DOCTOR OF PHILOSOPHY

Towards an Understanding of Communication within Pair Programming

Zarb, Mark

Award date:
2014

DOCTOR OF PHILOSOPHY

Towards an Understanding of Communication within Pair Programming

Mark Zarb

2014

University of Dundee

Towards an Understanding of

Communication within Pair Programming

Mark Zarb

Doctor of Philosophy

University of Dundee

2014

Acknowledgements

First and foremost, I would like to thank Dr Janet Hughes for her constant guidance, inspiring words and unwavering support; and Professor John Richards, for his invaluable encouragement and insightful comments on my work. I could not have asked for better supervision throughout the last three years.

Thanks go to my wife, Pamela: this thesis would not have been possible without your endless love, support and assistance. Thank you for always insisting I turn my frowns upside down… you were right, it does get better.

I would also like to acknowledge my friends – you were all individually instrumental in keeping me happy (and most importantly, fed) during the last three years. Thank you for listening, helping, and distracting when necessary.

Finally, thanks go to the many students and industry members who have helped along the way; and to the staff members at the School of Computing for their advice and assistance.

I would like to dedicate this work to my parents and to my family: your unconditional love and encouragement has shaped the direction of this work in more ways than you will ever know. Thank you.

The research work disclosed in this publication is funded by the Strategic Educational Pathways Scholarship (Malta). The scholarship is part-financed by the European Union – European Social Fund (ESF) under Operational Programme II – Cohesion Policy 2007-2013, "Empowering People for More Jobs and a Better Quality of Life".

Declaration of the Candidate

I declare that I am the author of this thesis; that all references cited have been consulted by me; that the work of which this thesis is a record has been done by myself; and that this thesis has not been previously accepted for a higher degree.

Mark Zarb

Declaration of the Supervisor

I declare that Mark Zarb has satisfied all the terms and conditions of the regulations under Ordinances 12 and 39, and has completed the required terms of research to qualify in submitting this thesis in application for the degree of Doctor of Philosophy.

Dr Janet Hughes

Table of Contents

Table of Figures

Table of Tables

Associated Publications and Awards

ZARB, M., HUGHES, J. & RICHARDS, J. 2014. Evaluating Industry-Inspired Pair Programming Communication Guidelines with Undergraduate Students. *Proceedings of the 45th ACM technical symposium on Computer science education (SIGCSE '14)*. Atlanta, GA, USA: ACM, 361-366.

ZARB, M., HUGHES, J. & RICHARDS, J. 2013. Industry-inspired Guidelines Improve Students' Pair Programming Communication. *Proceedings of the 18th ACM conference on Innovation and technology in computer science education (ITiCSE '13)*. Canterbury, England, UK: ACM, 135-140.

ZARB, M., HUGHES, J. & RICHARDS, J. 2012. Analysing Communication Trends in Pair Programming Using Grounded Theory. *Proceedings of the 26th BCS Conference on Human-Computer Interaction.* Birmingham, United Kingdom: British Computer Society.

ZARB, M. 2012. Developing a Coding Scheme for the Analysis of Expert Pair Programming Sessions. *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity (SPLASH '12): Student Research Competition.* Tucson, Arizona, USA: ACM, 237-238. **Second Prize at the Graduate Student Research Competition**.

ZARB, M. 2012. Understanding communication within pair programming. *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity: Doctoral Consortium (SPLASH '12)*. Tucson, Arizona, USA: ACM, 53-56.

The research discussed in this thesis was presented with the **Post Graduate Winner** award at the Agile Academic Awards, as part of the 2013 Agile Business Conference.

www.pairprogramming.co.uk

Abstract

Pair programming is a software development method which describes two programmers working together on the same computer, sharing one keyboard. This approach requires programmers to communicate frequently, which can lead the pair to experience certain benefits over solo programming, such as faster problem solving and a greater enjoyment of their work (Cockburn and Williams, 2001, Bryant et al., 2006). Many programmers approach their first pairing experience with scepticism, having doubts about their partner's working habits and programming style, and about the additional communication aspects that this programming style entails (Williams et al., 2000). Despite a significant amount of research into pair programming of over 15 years, it is not evident what communication between the pair contributes to the task of pair programming.

This work presents an analytic coding scheme which was derived from the observation of the communication of expert pairs working in industry. Over 35 hours of communication across 11 different pairs was analysed. This coding scheme was further refined to produce industry-inspired pair programming guidelines that assist novice pair programmers to improve their experience of pair communication.

Findings indicate that introducing these guidelines to novice student pairs can have a positive impact on their perception of intra-pair communication, and on their perception of their partner's contribution. Feedback received from expert pairs was used to add detail to the guidelines, which have been made publically available through an online resource.

Chapter 1: Introduction

1.1 Overview

Pair programming is a software development technique where two programmers work together side-by-side on the same machine to achieve their goals. This technique gained popularity in the early 2000s when it was presented as a key practice of the Extreme Programming software development methodology (Beck, 2000). An examination of the literature dates its use back to the early 1980s (Constantine, 1995), with empirical studies discussing the benefits of having two programmers dating back to 1993 (Wilson et al., 1993).

Many reported benefits of pair programming are reported for both novices and experts, including the pair experiencing a greater enjoyment of the work at hand, an increased knowledge distribution, and the production of better quality code (Cockburn and Williams, 2001). Further benefits are discussed when considering pair programming specifically in an educational context: students are more engaged in their collaboration, and seem more satisfied with their final work (Williams and Kessler, 2001).

In spite of these benefits, some developers are sceptical of their first pair experience and of its promised collaborative value (Williams and Kessler, 2000), citing doubts about their partner's work habits and the added communication demands that this style of programming requires. *Communication* itself is frequently cited as a common barrier to pair programming by novices (e.g. Cockburn and Williams, 2001, Begel and Nagappan, 2008). However, if a pair does not communicate, they are not pair programming, but effectively they are only reviewing each other's code. Within the literature, it can be seen that communication is not only an integral contributor to the success of pair

programming, but also one of the main causes of its failure (Sanders, 2002, Begel and Nagappan, 2008, Murphy et al., 2010).

This thesis presents research which investigates common communication patterns and trends displayed by expert pairs of programmers. This allows for an understanding of how intra-pair communication is structured. This knowledge is then cast into guidelines and examples which could be used to assist novice pair programmers in learning to communicate more effectively when working together.

1.2 Aim of the Thesis

Novice pair programmers find communication within their pairs to be one of the greatest difficulties they face when starting to pair program (Williams and Kessler, 2000, Sanders, 2002). However, pairs cannot program without exhibiting a certain amount of communication: "*Effective pairs chatter; silence is a danger signal*" (Williams and Kessler, 2002).

The research question is identified following a review of the existing literature in the field as shown in Chapter 2:

> ***Can extracted communication patterns from expert pair programmers be used to help novice student pairs to improve their intra-pair communication?***

This research question is answered gradually throughout the course of the thesis, culminating in a discussion in Chapter 8.

Following a literature review in Chapter 2, the thesis describes the development of an analytic coding scheme derived from the observation of one expert pair working in industry. This is used to observe and analyse a number of instances of expert pair communication across different expert pairs working in different sectors of industry.

Usage data for the coding scheme is analysed and is used to identify patterns of verbal communication that are observed across multiple pairs within the industry. These patterns are cast into pair programming guidelines, with the aim of aiding novice pairs in their communication. Novice pair programmers are observed and exposed to the industry-inspired guidelines and interviewed, in order to understand what impact the guidelines have on their perception of communication within pair programming. Further evaluations are carried out in order to determine what effect the guidelines have on the novice pair's communication. The pair's communication effectiveness is evaluated by analysing the individual's perception of their partner's contribution, as well as their self-reported ease of communication experienced during the evaluation session.

Finally, feedback is collected from both novice learners and expert developers in order to gain an understanding of how the guidelines are used, and to inform the next stages of research.

1.3 Contributions to Knowledge

Despite the amount of research which looks into pair programming, discussed in Chapter 2, it is not fully clear what the communication within the pair contributes, with few studies investigating this field in detail (Stapel et al., 2010). If this contribution can be understood, it would lead to improved teaching practices for pair programming, and could help identify obstacles to successful pairing in industrial settings.

The main contributions to knowledge of this thesis are:

(i) A coding scheme has been identified that can be applied to analyse pair programming communication. The codes were derived from observing and examining expert pair communication;

(ii) Pair programming guidelines have been created based on the application of the coding scheme to identify patterns of communication;

(iii) The guidelines have been evaluated with student pairs. This showed that exposure to these guidelines improved the self-perceived communication experience of novice pair programmers, but had no significant impact on their success levels.

1.4 Thesis Structure

This thesis consists of eight chapters. Following the introduction, **Chapter Two** provides an overview of the literature, starting by outlining the pair programming methodology, as well as its benefits and drawbacks for experts and novices - in particular, student novices. This highlights communication as a common pitfall. Several papers discussing the observation of communication in a pair programming environment are then discussed. This is followed by a review of qualitative methodologies that could be used to run exploratory and investigative studies on communication with experts. This chapter finishes with the research question for this thesis.

Chapter Three presents an informative study in which several hours of communication from one expert pair are observed and analysed. This in-depth analysis leads to the development of several analytic codes and patterns which are verified against other industry-based pairs in **Chapter Four**. This leads to the creation of the pair programming guidelines which embody the knowledge gained from observing these expert pairs.

Chapter Five presents an exploratory study with novices to pair programming. A class of undergraduates are introduced to pair programming. Following this, a subset of this group is exposed to the industry-inspired guidelines for several weeks. Interviews are held to determine whether students applied the guidelines within their own pairs and whether they found the guidelines to be beneficial. Results are presented and discussed.

Chapter Six documents a series of evaluations, aiming to explore the potential benefits of exposing novice pairs to the industry-inspired guidelines. Evaluations indicate that when compared to a control group, pairs who were exposed to the guidelines reported a greater ease of communication within their pair, and also reported perceived improvements on their partner's contribution.

In **Chapter Seven**, feedback is collected from both novice and expert pairs with regards to the guidelines. Comments from industry-based experts are summarised and used to add further guidelines to the original.

The thesis concludes in **Chapter Eight** with a summary of the research, where the research question posed for this thesis is re-visited and discussed. This is followed by the thesis contributions, and the thesis output. The chapter ends with an outline of proposed directions for future research in the field.

Chapter 2: Literature Review

This chapter presents a literature review on pair programming. It considers the importance of intra-pair communication, the use of pair programming in industry and in academic settings, and the use of qualitative research methodologies that can aid in the observation of communication within pair programming.

The review of literature begins with an introduction to pair programming, and a discussion on communication (both verbal and non-verbal) and its role in pair programming. This is followed by an identification of benefits and issues in the context of pair programming within both industry and academic settings. *Communication* is presented as a prevalent issue, and previous studies are discussed in order to understand how researchers have observed and studied communication within the context of pair programming. Grounded theory is then presented as a suitable methodology for the next stages of this research. The chapter concludes with a discussion of the identified gap in the literature, thus leading to the research question that will be explored throughout this thesis.

2.1 Introduction to Pair Programming

Williams and Kessler (2002) describe pair programming as a coding activity, during which two developers collaborate continuously on the same program, usually at the same computer. The members of the pair each take on different roles: the *driver* has full control of the keyboard, while the *navigator* is in charge of reviewing the code and performing continuous analysis (Williams and Kessler, 2000). It is common practice for partners to switch roles frequently, usually at agreed intervals, for example, following completion of a method or unit test or after a set period of time. Due to the nature of pair programming, *communication* should occur continuously: effective communication

is a necessity for pair programming success (Begel and Nagappan, 2008, Sharp and Robinson, 2010).

Pair programming has been practised and advocated for many years: Wilson et al. (1993) performed one of the earliest empirical studies that indicates benefits of students pairing on programming tasks. A publication in 1995 by Constantine reports the observation of pairs of programmers, termed "dynamic duos" in the early 1980s, noting that the pairs produced code faster and with fewer errors than their solo counterparts. The procedure described here involved one programmer writing code, and the other peering over their shoulders (Constantine, 1995).

Coplien (1995) published an organisational pattern termed *Developing in Pairs*. This pattern targeted the issue that "some problems are bigger than any one individual", and the solution described was to "pair compatible designers to work together; together they can produce more than the sum of the two individually" (Coplien, 1995).

The emergence of the Extreme Programming software development methodology (XP) "that favours both informal and immediate communication over the detailed and specific work products required by any number of traditional design methods" (Beck, 2000) introduced the pair programming practice to the general programming community. The XP methodology was initially defined as consisting of 12 key practices, one of which was pair programming (Williams et al., 2000, Beck, 2000). Following five years of experience and research, XP was re-defined: its original key practices were divided into primary practices (useful practices independent of the development methodology being used) and corollary practices (which should not be implemented before a core set of primary practices are put in place) (Beck and Andres,

2004). Some of the initial twelve key practices were relegated to corollary practices – but pair programming remained as a primary practice of the methodology.

XP, together with other 'alternative' methodologies to the more traditional waterfall-style development, was incorporated into the Manifesto for Agile Software Development[1]. This set of principles was to encompass "better ways of developing software", emphasising items such as individuals and interactions, and customer collaboration (Cohen et al., 2004). To date, pair programming is one of the most documented and most popular agile process (Hannay et al., 2009, Dybå et al., 2012).

In 2003, a report by Cusumano et al. shows that 35.3% of 104 surveyed software development companies worldwide were using pair programming. More recently, Chong and Hurlbutt (2007) write that "more and more commercial companies are considering its use", and Domino et al. (2007) stated that it is "gaining organisational interest", with large companies reportedly using pair programming. Furthermore, Salleh (2008) states that "the practice of pair programming has been widely implemented in the industry".

2.1.1 Guidelines for Implementing Pair Programming

By definition, a *guideline* is a general rule or a piece of advice, synonymous with a recommendation or a suggestion. Some researchers and instructors have presented their experience with teaching pair programming as guidelines for implementation; these guidelines will be discussed in this section.

Bevan et al. (2002) observe that the structure of the class can fail to encourage a consistent pair programming environment. They therefore present a number of

[1] http://www.agilemanifesto.org/

guidelines to be used as a framework by instructors interested in adopting pair programming. Some of these guidelines belong under headings such as: *pair within sections, pair by skill level, institute a coding standard* and *create a pairing-oriented culture*. Similarly, Williams et al. (2008) draw on over seven years of teaching experience in order to establish eleven guidelines for classroom management when pair programming is being used. These guidelines, like Bevan et al.'s, are aimed towards instructors, providing additional support on the points such as the following: *supervised pairing experience, teaching staff pair management, balancing individual and collaborative work,* and *pair programming ergonomics*.

With regards to student-based guidelines, several authors make reference to giving students a paper by Williams and Kessler (2000) as "guidelines to introduce the pair programming concepts" (McDowell et al., 2003, VanDeGrift, 2004, Mendes et al., 2005). The paper, "All I Really Need to Know about Pair Programming I Learned in Kindergarten", describes the basics of pair programming under headings such as *share everything, play fair, don't hit your partner, put things back where they belong*, etc. In a separate paper, Williams et al. (2000) refer to these headings as "guidelines for transitioning from solo to pair programming".

The discussed papers present sets of guidelines targeted towards solo programmers or instructors, but none mention the process used to create the guidelines, with each paper drawing on the respective authors' observations and experiences to inform and create the guidelines.

2.2 Communication and Pair Programming

Pair programming is a highly communication-intensive process, consisting of both verbal and non-verbal forms of communication (Sharp and Robinson, 2010). Williams

and Kessler (2002) write that effective communication within a pair is paramount, and that lengthy periods of silence within the pair should be considered a danger signal. Furthermore, several studies, both in industry and in academia have concluded that apparent successes of pair programming are due to the amount of verbalisation that this style of coding requires (Chong and Hurlbutt, 2007, Freudenberg et al., 2007, Hannay et al., 2009).

An experiment conducted by Bryant et al. (2006) shows that in expert pairs, the communication distribution between the driver and the navigator is 60:40 respectively. After analysing 23 hours of dialogue produced by pair programmers, the researchers conclude that "the benefits attributed to pair programming may well be due to the collaborative manner in which tasks are performed" (Bryant et al., 2006).

Watzlawick et al. (1967) consider that within professional relationships there is a necessity for constant communication. This necessity can be seen within pair programming: Flor and Hutchins (1991) observe that the exchange of ideas, feedback, and constant debate – thus, communication – between two programmers collaborating on a software maintenance task significantly reduced the probability of ending up with a poor design. Wilson (1993) shows that in an academic context, collaborative work benefits problem-solving efforts: teams that were allowed to communicate whilst working on a software development task were seen to have a higher confidence in their solution.

Industrial developers surveyed by Begel and Nagappan (2008) define a prospective partner with good communication as one who embodies the following qualities:

- A good listener;

- Articulate;
- Easy to discuss code with;
- Very verbal, to make the thought process easy to understand;
- Enjoys debating and discussing code;
- Asks questions, and provides opinions.

Communication is considered to be a "vital aspect of pair programming" (Lindvall et al., 2002), while Beck (2000) writes that it is "the first value of pair programming", and that coding standards should emphasise communication. Aiken (2004) reports that when pair programming, "no more than a minute should pass without verbal communication". The area of communication within pair programming is seen as an important topic of research interest (Stapel et al., 2010), and it is also considered "one of the most important factors" within software engineering (Gallis et al., 2003). Furthermore, surveyed developers at Microsoft have rated 'good communication skills' as being a top attribute for good pair programming partners (Begel and Nagappan, 2008), and it is seen as an "integral" concept for agile methodologies as it helps people to work better when partnered (Cockburn and Williams, 2001, Nawrocki and Wojciechowski, 2001).

Choi et al. (2009) found that there is no correlation between communication and satisfaction, compatibility, or confidence. Pairs who exhibited a high level of communication within the pair did not necessarily experience a high level of satisfaction, compatibility between partners, or a high level of confidence regarding the finished product. Sfetsos et al. (2006) found that for a group of pair programming students, there was a significant positive correlation between the number of

communication transactions within the pair and the pair's productivity. These two statements are contradictory, but serve to show that despite existing research showing that communication is intrinsic to the pair programming process, different authors are using different measures (e.g. satisfaction, productivity, compatibility) to understand communication within the pair.

Freudenberg et al. (2007) write: "the cognitive aspects of pair programming are seldom investigated and little understood". In one study examining communication per se, Stapel et al., (2010) hypothesise that there could be a difference in the rate of communication between novice pair programmers (defined as "new to pair programming and unfamiliar with each other") and professional ones (not explicitly defined in Stapel et al.'s paper, but used to indicate "experienced" pairs from discussed studies; i.e. industry-based pairs). The authors believe that this is due to the fact that a more experienced and confident pair will probably be more at ease with communicating and sharing ideas, whereas a more novice pair may be concerned about repercussions to sharing the wrong idea. The communication (or lack thereof) within a pair might determine the success of a pair programming exercise: if the pair does not communicate, then the programmers are only reviewing each other's code (Gallis et al., 2003).

2.2.1 **Verbal and Non-Verbal Communication**

When working in a pair, programmers are expected to collaborate both verbally and non-verbally (e.g. by using gestures, or certain facial expressions to express emotion). An initial observation reports on both these styles of communication, and is given in Chapter 3 of this thesis.

Spoken data has been an important element of many experiments in computer science education and software engineering research. Verbal data collection is frequently facilitated by observing groups of individuals working together (Murphy et al., 2010). In a pair programming context, Bryant (2004) comments that verbal communication is "natural", and "absolutely essential". A literature review by Hughes and Parkes (2003) and subsequent reporting by Freudenberg et al. (2007) indicates that the analysis of verbalisation may be a useful method for use in the study of pair programmers, so that real-time insight about the knowledge that the subjects use "can be formally mapped, rather than speculated about." More recently, Stapel et al. (2010) report on a study where verbal communication is used to better understand the communication structure exhibited by pairs who are programming together.

Non-verbal communication is also present in pair programming; for example, "a developer can contribute by using external representations or by pointing on the screen" (Plonka et al., 2012). When asking industry members and students to rank their preferred personality traits in a potential pairing partner, Chao and Atli (2006) show that paying close attention to non-verbal cues was highly ranked by both groups. Whilst several studies mention non-verbal communication and point to its importance in pair programming (e.g. in their study limitations, Freudenberg et al. (2007) acknowledge that despite the study's focus on verbal communication, there are other, non-verbal means of communication that were excluded from the study: "for example [..] when they used particular facial expressions or gestures"), there seems to be little work done on the analysis of this type of communication. This could be due to the fact that this type of analysis can be difficult to interpret due to its ambiguity: discerning between actions and meanings can be complex (Pearson et al., 2006).

An analysis of the literature was carried out with the aim of understanding the types of communication considered by various authors. A sample of fifteen papers was selected to cover a range of years and sources, and each paper was re-examined to identify the various types of communication discussed (e.g. verbal or non-verbal). The results are presented in Table 1 below.

Table 1: Types of Communication in Existing Studies

Author	**Type of Communication Discussed**
Gittins et al. (2001)	Unspecified
Cockburn and Williams (2001)	Verbal
Lindvall et al. (2002)	Unspecified
Gallis et al. (2003)	Both verbal and non-verbal
Aiken (2004)	Verbal
Ally et al. (2005)	Unspecified
Chao and Atli (2006)	Both verbal and non-verbal
Freudenberg et al. (2007)	Both verbal and non-verbal
Chong and Hurlbutt (2007)	Verbal
Begel and Nagappan (2008)	Both verbal and non-verbal
Hannay et al. (2009)	Verbal
Stapel et al. (2010)	Verbal
Murphy et al. (2010)	Verbal
Sharp and Robinson (2010)	Both verbal and non-verbal
Plonka et al. (2012)	Both verbal and non-verbal

Five of these studies focused solely on verbal communication ("apparent successes of pair programming [are due] to the sheer amount of verbalisation" (Hannay et al., 2009) / "communication within the pair occurred chiefly through conversation (Chong and

Hurlbutt, 2007), with none of the sampled papers focusing solely on non-verbal communication. Interestingly, two papers do not specify what kind of communication is being discussed or observed. Eight papers acknowledge both verbal *and* non-verbal communication: Gallis et al. (2003) say that developers need to communicate *both* verbally and non-verbally in order to truly pair program, and Begel and Nagappan (2008) describe a desired pairing partner as being "very verbal", and also acknowledge that excellent communication is both verbal and present in the developer's body language.

It can be seen that "the process of pairing is fundamentally about communication – both verbal and non-verbal" (Sharp and Robinson, 2010). Most of the surveyed literature acknowledges both types of communication; however, more in-depth analysis (e.g. as reported in section 2.5.1) focuses mostly on verbal communication. Sharp and Robinson (2010) acknowledge this by stating that "communication in agile development is [..] predominantly verbal".

Whilst the importance of non-verbal communication is acknowledged in this thesis, the work discussed in this thesis will firstly address *verbal* communication. When considering the surveyed literature, it can be seen that in agile development, previous research focusses mostly on verbal communication; furthermore, it is expected that it will be more natural for pairs to implement verbal communication patterns in their interactions. The analysis of *non-verbal* communication is then considered as further work in Chapter 8.

2.2.1.1 Silent Pauses in Verbal Communication

Campione and Véronis (2002) present a study on pause duration in verbal communication, denoting each 'pause' to be a short gap in speech of up to 2000ms. In the study, they analyse 6000 pauses in speech, with participants either reading ('read speech') or conversing naturally ('spontaneous speech'). The authors show that the distribution of pause durations is observed to be strongly skewed to the left, as replicated in Figure 1:

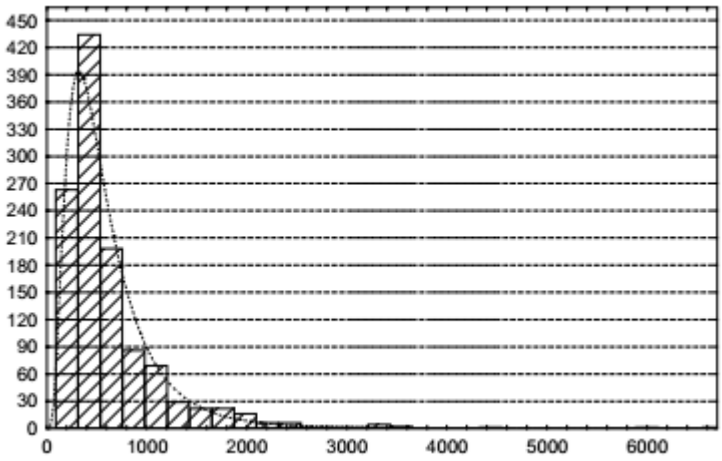


Figure 1: Distribution of pause durations (Campione and Véronis, 2002)

The distribution in Figure 1 shows that most pauses in spontaneous speech were observed to last for approximately 500ms, when considering the highest peak in the chart. To quantitatively summarise this data and get a more accurate figure, the authors comment that: "the arithmetic mean (629ms) is not a reliable measure of central tendency, given the strong skewness of the distribution. Much more stability is observed when medians (451ms) are used" (Campione and Véronis, 2002).

2.3 Pair Programming Benefits

Pair programming is widely used in industry (Domino et al., 2007, Salleh, 2008) and in academia (Katira et al., 2005), where it is typically introduced in tertiary education. In both contexts, pair programming encourages programmers to talk to each other and to themselves – this 'pair pressure' adds benefits such as greater enjoyment and increased knowledge distribution (Williams and Kessler, 2001, Bryant et al., 2006). Benefits of this approach to programming have been investigated through controlled experiments in areas such as cost reduction, continuous review and programmer satisfaction.

2.3.1 Benefits Reported in Industry

Arisholm et al. (2007) performed one of the larger-scale experiments, during which the authors ran pair programming studies with 295 industry professionals from Norway, Sweden and the UK, divided into 98 pairs and 99 individuals. As part of their analysis, they looked at differences in performance between pair programmers and single developers, taking note of correct solutions achieved, and the time taken to do so. The results described show certain benefits to pair programming over working as a single developer. The authors show that on complex systems, the pairs achieved 48% more correct solutions with no significant time difference when compared to single developers. When working on simpler systems, the pairs were 20% quicker to achieve completion, with no significant difference in correct solutions.

Many developers are initially sceptical of the value of collaboration that pair programming seems to promise, as they do not expect to gain any benefit from the experience (Williams and Kessler, 2000). In their work, Williams and Kessler discuss a survey where 91% of the programmers questioned indicate an agreement that their pair

partner's "buy-in" to the experience was a critical component to pair programming success, as it helped alleviate the initial scepticism within the pair.

Cockburn and Williams (2001) present a list of benefits to adopting a pair programming approach. These are shown in the following bullet points, each accompanied by a number of studies which verify these benefits.

- *When pair programming, many mistakes are noticed as they are being typed, rather than relying on quality assurance tests at a later stage in the development process. When compared to solo programmers, pairs have fewer errors in their code, and typically consider more design alternatives to the problem at hand, thus producing simpler designs (Cockburn and Williams, 2001).*

Jensen (2003) discusses a case study detailing the introduction of pair programming into a team of ten programmers with a wide range of experience. A productivity gain of 127% was reported after using pair programming, with an error rate that was "significantly less than normal" for the organisation. Following the introduction of pair programming within their organisation, Pandey et al. (2003) report "the best example of productivity improvement in the entire department". Furthermore, working in pairs made the developers feel that they were contributing to and considering more design alternatives, ultimately feeling that the solution the pair chose to implement was the best solution.

A case study reported by Vanhanen and Korpi (2007) shows that developers consider pair programming to be a contributing factor for the resulting low defect counts in the system, and to be especially useful for more complex tasks. In a separate case study, di Bella et al. (2012) report that during a data collection period of 14 months, fewer

defects were introduced into the code when pair programming was being practised by the developers.

Dybå et al. (2007) and Hannay et al. (2009) present a systematic review and a meta-analysis of existing studies within the literature, discussing the use and adoption of pair programming. The general consensus across the various studies is that pair programming leads to an increased quality of software, and that it is beneficial for achieving correctness on programming tasks.

Ally et al. (2005) state that engaging in pair programming can help to improve communication skills in the team and therefore, improves the overall team's interaction. Williams and Kessler (2002) suggest that developers who are pair programming use communication to "show their colleagues what they are working on; to look at what their colleagues are doing and to see what they can learn; to spot as many loopholes, flaws and mistakes as possible."

- *Upon completion of a pair programming project, multiple people are able to understand more parts of the system, as opposed to traditional approaches, where one person is solely responsible for large parts of the system (Cockburn and Williams, 2001).*

Luck (2004) states that if two people are involved with the design of code, "collective ownership is enhanced". Vanhanen and Korpi (2007) report that developers involved in pair programming teams generally had higher involvement in more parts of the system than solo developers, and as such, "all developers considered that pair programming increased their knowledge of the system more than solo programming". A survey run by Begel and Nagappan (2008), sampling responses from 487 contributors, shows that one

of the top perceived benefits of pair programming was "spreads code understanding". This is also seen in Fronza et al. (2009) who, following a 10-month study, show that pair programming helps spread code understanding across developers, and therefore facilitates knowledge transfer.

- *The pair indicates that they have a greater enjoyment of their work whilst pair programming (Cockburn and Williams, 2001).*

By analysing survey results for 108 developers, Succi et al. (2002) show that pair programming has a significant and positive influence on the developers' satisfaction due to factors such as "increased communications, speed of communication of design changes, and organisation of meetings". Furthermore, Luck (2004) observes that "developers seem to really enjoy the flexibility of pairing".

The benefits of following a pair programming methodology are not restricted to the code quality. Some researchers argue that pair programmers experience fewer interruptions when compared to single developers.

Williams and Kessler (2002) indicate that it is not common to be interrupted by people who are not part of the pair: "[Other developers] see us already working with someone else, and they leave us alone. The net effect is that we have bigger blocks of uninterrupted time, which is good for our mental state and our progress."

This claim was verified by Chong and Siino (2006), who ran ethnographic studies to compare interruptions between two teams of developers: one team consisted of pair programmers, and the other consisted of developers working 'solo'. It was reported that the interruptions that occurred within the pair programming team were consistently shorter than the interruptions for the 'solo' developer's team, regardless of interruption

type and source. Furthermore, it was observed that developers in the pair programming team waited for a suitable moment before interrupting their pairing co-workers.

The authors suggest that as pair programming is a highly cooperative methodology, pairs may feel a sense of strong social obligation to their pair partners, which allows them to handle interruptions quickly, so that the interrupted pair can quickly return to their primary task.

Plonka et al. (2012) describe several factors that can lead to a pair member losing focus and leaving their partner to work by themselves, thus leading to the pair becoming disengaged. These factors include external interruptions, time pressures and the pair being mismatched, leading to social pressure where the novice member of the pair was not comfortable challenging the more senior member.

An ethnographic study reported by Chong and Hurlbutt (2007) discusses pair behaviour in relation to each developer's individual expertise within the pair. When both members of the pair had equivalent levels of expertise, they were engaged equally in programming activities. However, when the distribution of expertise differed, the programmer with more experience seemed to dominate the interaction. The authors discuss the interaction between several programmers, as per the excerpt below:

> "*Ilya dominates the interaction, determining how and what to implement while Hugh takes directives (to the keystroke) from him; Hugh primarily asks for minor clarifications. Hugh's level of participation here is actually unusually low (he will, in fact, begin to contribute somewhat more actively later in the session), but the structure of this exchange is consistent with the majority of the pair programming interactions on the team as a whole: the programmer with*

greater task knowledge or code base familiarity dominated. This occurred regardless of which programmer was at the keyboard."

2.3.2 Benefits Reported in Academic Settings

"Pair programming shows students that being in computer science is about an intensive social experience, and that learning and performance in computer science is made better by working with others." (Porter et al., 2013)

Katira et al. (2005) state that pair programming is "used widely in software engineering education". The following studies depict and discuss students in an educational context experiencing pair programming for the first time: these students are considered to be pair programming novices.

Within the classroom, pair programming is seen to be valuable (Williams and Kessler, 2002, Begel and Nagappan, 2008, Hanks, 2006). Its use in educational settings has reported usage in the United States, the United Kingdom, Germany, New Zealand, India and Thailand (Hanks et al., 2011). Students working in pairs are seen to be more satisfied, solve problems faster than non-paired students, and have improved team effectiveness (Williams et al., 2000, McDowell et al., 2003, Srikanth et al., 2004). Pair programming among students is not a deterrent to individual student performance (Johnson and Caristi, 2001): pairing students were shown to be more likely to complete courses related to computer science and achieve a successful grade for their assignments when compared with their solo counterparts, as well as gaining an improved comprehension of unfamiliar topics (Williams et al., 2002, Nagappan et al., 2003a, Braught et al., 2008). Students who were exposed to pair programming in the classroom reported that having a partner with whom to discuss unfamiliar topics was helpful

(Cliburn, 2003), and that this improved their comprehension of unfamiliar topics (Kavitha and Ahmed, 2013). Interestingly, Hanks (2007) shows that paired students experienced the same problems and struggles encountered by solo students, despite the benefits afforded by a pairing approach. Similar results were reported by Porter et al. (2013): paired students had a higher pass rate than their solo peers, and were more likely to continue on the next course.

Initial observations with student programmers learning to work in pairs reveals several benefits to this approach (Williams and Kessler, 2001, Werner et al., 2004). Students working in pairs answer each other's questions, rather than considering their instructor as the only source of advice – which contributes to the students' learning process. The use of pair programming, and the subsequent 'pair pressure', causes students to work on projects earlier and to budget their time more wisely. Pair programmers took less time to complete set tasks (DeClue, 2003), and programs produced by paired students were seen to be significantly better than programs produced by individual student programmers. Students surveyed by Sanders (2002) following an initial experience of pair programming reported on experiencing a skewed perception of time, in which they felt they worked for less time than they actually did.

Williams et al. (2002) show that student pairs displayed a higher confidence and a more positive attitude in their project work when compared to solo student developers. Furthermore, pair programming has been proven to be useful in a learning environment for solving problems and complex tasks, and finding mistakes in simple code segments (Hulkko and Abrahamsson, 2005, Williams and Kessler, 2002). Programming students agree that they have more confidence in their final solution when it is achieved through pair programming (Williams and Kessler, 2000), and perceived pair programming as

being valuable to their learning (VanDeGrift, 2004). The process of pair programming leads to students who are more satisfied with their work regardless of their ability and grade-level (Kavitha and Ahmed, 2013, Vanhanen and Lassenius, 2005), and who are more self-sufficient. The student perception of pair programming on various tasks was examined by Chaparro et al. (2005), who reported that when considering program comprehension, refactoring and debugging, students were effective across all three.

Observations with undergraduate student pairs suggested that students who communicated within their pair more frequently were seen to attempt to solve more problems (Murphy et al., 2010). Stapel et al. (2010) have discussed two benefits that occur as a product of the communication that occurs within a pair programming environment:

- More learning takes place when technical experience is shared within the pair. This is of high relevance, as "better educated developers are more likely to produce high quality code".
- Team building occurs when the pair is collaborating closely – whether they are communicating about the task at hand, or about off-topic issues. This communication allows the pair to establish a common context, which simplifies future communication.

DeClue (2003), McDowell et al. (2003) and Srikanth et al. (2004) noted that paired students change their partners frequently in class, as well as changing their designated role within the pair. These studies show that it is not optimal for a student to work with the same partner over a lengthy period of time (e.g. an entire semester). By changing partners frequently, students are exposed to more classmates and more ways of

working, therefore learning a variety of ways to solve potential communication problems. Srikanth et al. (2004) present comments from educators and students which show that frequent pair rotation allowed for each student to have multiple sources of feedback during peer evaluations and helped the students be exposed to new ways of learning by collaborating with different classmates. Frequent rotation also provided an easier way to handle the more dysfunctional pairs within the class.

Sanders (2002) ran a pilot experiment where students were exposed to pair programming after being asked to write opinion papers regarding its use. After having experienced pair programming for the first time, students were then asked their opinion of the process. The students noticed several immediate benefits to pair programming which are similar to the various benefits discussed in the previous paragraphs: for example, pairs of unequal abilities found that the weaker student was able to learn more by talking to the stronger student, and vice-versa; the stronger students improved their understanding of topics learnt by discussing them with the weaker students. Several students also reported improved relations with their partner and more efficient problem avoidance.

Hanks (2007) indicates that pair programming is an effective tool that allows pairing students to resolve problems quicker and more often than solo students. The advantages of paired students are also documented by Lui and Chan (2006). They demonstrate that pairs of novices displayed significant productivity gains when compared to solo novices. However, no significant differences were reported when comparing expert pairs to solo expert developers, suggesting that pair programming is of greater learning benefit to novice developers than expert ones.

Students seem to prefer to pair with someone they perceive to be of similar technical competence (Williams et al., 2006) – when a weaker student is paired with a stronger student (therefore creating a 'novice-expert' pair in this context), the stronger student tends to take over, leading the weaker student to be largely an observer instead of a participant (Melnik and Maurer, 2002, Braught et al., 2010).

Several studies show that matching pairs based on skill level is beneficial for productivity and for students' self-confidence (Melnik and Maurer, 2002, Sanders, 2002, Bevan et al., 2002, Nagappan et al., 2003b, Katira et al., 2004, Begel and Nagappan, 2008, Braught et al., 2010). When both members of a student pair were observed to have equivalent expertise or skill levels, each member became more involved in the programming activities at hand, thereby producing "their best work" (Thomas et al., 2003). Furthermore, several studies show that matching pairs based on skill level is beneficial for their overall productivity (Melnik and Maurer, 2002, Sanders, 2002).

2.3.3 **Summarised Benefits**

The benefits reported above are presented in separate contexts: benefits reported through studies carried out in industry, and benefits reported through studies carried out in education. It can be seen, however, that some of these benefits are common to both contexts:

Table 2: Common Benefits of Pair Programming

Benefit	**Reported in Industry**	**Reported in Academic Contexts**
The pair produces better work.	"the pair programmers had an 48% increase in the proportion of correct solutions" (Arisholm et al., 2007)	"students who performed in pairs outperformed those who worked alone" (McDowell et al., 2003)
The pair produces fewer errors.	"the number of errors [..] was significantly less than normal" (Jensen, 2003)	"students felt the presence of a partner helped complete the assignments [..] with fewer errors" (DeClue, 2003)
There is increased enjoyment.	"96% agreed that they enjoy their job more when programming in pairs" (Williams and Kessler, 2001)	"paired students enjoyed working on their assignments more than non-paired students" (McDowell et al., 2006)

2.4 Pair Programming Issues

Whilst there are several benefits to pair programming as reported above, a range of concerns still needs to be studied and addressed. These concerns will be discussed in the following section.

Pair programming is an agile technique, and forms part of Extreme Programming. Beck (2000) states that communication is one of the four most important values which developers should consider when adopting Extreme Programming. Moreover, Lindvall et al. (2002) assert that one of the most important success factors for agile is support for rapid communication. The publications listed in this section, as well as the discussion in section 2.2 of this thesis, present evidence showing that effective communication within the pair is intrinsic to pair programming and an important factor to consider when

considering pair programming success. In the following section, the topic of communication as an issue of pair programming will be discussed in further detail.

2.4.1 Issues Reported in Industry

The ‘writing code’ component of programming has traditionally been practised as a solitary activity. When working in a pair, members of the pair are typically required to physically work together and share responsibilities. Generally, the driver expects the navigator to point out flaws in the code and give direction, but this could give programmers a sense of discomfort, leading to a lack of productivity (Cockburn and Williams, 2001). Hence, experienced programmers are sometimes reluctant to program with another person, as they feel that their code is “personal”, or that they might be slowed down by their partner.

Pair programmers enter a pairing session expecting certain attributes from their partner (Begel and Nagappan, 2008): a good partner should communicate well, complement the other’s skills, and complement the other’s personality. *Disagreements* and *bad communication* are both placed in a ‘top 10’ list of pair programming problems as perceived by engineers surveyed at Microsoft:

> “Pairs find it hard to get a consensus in ideas. *‘Sometimes we waste time on discussion’*, where we surmise [..] that to many respondents, ‘discussion’ is synonymous with ‘argument’.” (Begel and Nagappan, 2008)

Furthermore, a good partner should “be more experienced in areas that [the developer is not]”, and be helpful when solving problems. As a pair, the team is expected to be fast, efficient, effective, communicate well and work without irritating each other. Begel and Nagappan (2008) show that high levels of expectation can lead to developers being

anxious about the pairing process; this may make developers enter a pair programming situation with caution and apprehension. Many programmers therefore approach their first pair programming experience with a sense of scepticism, having doubts about their pair partner's working habits and programming style, about disagreeing on the implementation process, and about the added communication aspects that this style of programming entails (Williams et al., 2000, Greene, 2004).

2.4.2 Issues Reported in Academic Settings

The main issue reported in academia is that students find scheduling time for meetings to be particularly difficult, due to conflicting schedules: pairs of students typically had different timetables and found it difficult to find time for the pair to be able to work together. A report by VanDeGrift (2004) describes students being asked to list their perceived disadvantages of pair programming. Answers from 293 students within a large university were categorised, with the following being the highest reported disadvantages of pair programming:

- Scheduling time for meetings (47.8%);
- Partner personality differences (25.6%);
- Partner skill level differences (22.9%).

A number of studies have reported findings similar to the above in various different educational settings (Simon and Hanks, 2008): for example, Cliburn (2003) and Srikanth et al. (2004) show that pair programming could lead to several issues to do with scheduling, pair incompatibility and unequal participants. Moreover, Thomas et al. (2003) indicate that their less successful paired students mentioned being frustrated, guilty, and feeling like they had wasted their time. This pair incompatibility can be a

great cause of concern for students, with studies showing that amongst groups of students, those with a higher skill level report the least satisfaction when paired with students who are less skilled (Thomas et al., 2003).

Melnik and Maurer (2002) discuss various issues observed in the classroom, noting that some students found the pairing element particularly difficult, and "could not trust other people's code". Furthermore, approximately 50% of first-time student pair programmers reported that the various forms of difficulties within the pair contributed to *communication* being the main problem with the pair programming process (Sanders, 2002).

Despite the centrality of communication referenced in the literature above, there are practical issues with pair programming communication. A number of these have been briefly referenced earlier in this chapter: the communicative collaboration required by pair programming can cause discomfort for both the driver and the navigator (Cockburn and Williams, 2001), leading to reduced communication effectiveness and lower productivity (Aiken, 2004).

2.4.3 **Summarised Issues**

The issues reported above are presented in two contexts: issues reported from industry and issues reported from the classroom. Several issues are seen to be common to both contexts.

Table 3: Common Issues of Pair Programming

Issue	**Reported in Industry**	**Reported in Academic Contexts**
Individuals have concerns about the pairing process due to the communication that is expected of them.	"many venture into their first pair programming experience sceptical that they would benefit from collaborative work [and] about the added communication that will be required" (Williams et al., 2000)	"roughly 50% of the students also reported various forms of communication difficulties" (Sanders, 2002)
There is a reluctance to share ideas with a partner.	"programmers initially resist pair programming [due to] a reluctance to share ideas" (Ally et al., 2005)	"[students] can be reluctant to share solutions they may have taken a long time to devise" (Cleland and Mann, 2003)
There can be scheduling conflicts.	"working with a partner will cause trouble coordinating work times" (Cockburn and Williams, 2001)	47.78% of surveyed students indicated that "scheduling time for meetings" was a disadvantage of pair programming (VanDeGrift, 2004).

Whilst this communication is an important and essential aspect of pair programming, it is also an issue and a barrier for first-time pair programmers in both industrial (Williams et al., 2000, Begel and Nagappan, 2008) and academic (Sanders, 2002) contexts. The literature posits interesting questions about communication, but ultimately it can be seen that many authors simply view communication as the essence of paired programming, and as a result, do not investigate how communication happens within pair and how it is or is not effective (Stapel et al., 2010, Sharp and Robinson, 2010).

2.5 Researching Pair Programming Communication

A better understanding of pair communication should lead to increased knowledge of implications that this communication has on the pair's effectiveness and success. It is expected that any study of pair communication will invariably require the observation of a pair; specifically, the observation of the way a pair communicates.

The following sections will describe existing studies in the literature where pair communication is observed. Different methodologies that can be used in a research-based study are then identified, leading to the selection of grounded theory as the main research methodology for the next stages of this research.

2.5.1 Existing Studies

There is little research aimed at understanding the detailed nature of communication in pair programming (Höfer, 2008). This section will discuss existing studies within this research thread, considering studies which were primarily observation-based, studies which generated ways to calculate metrics, and studies through which coding schemes were derived.

2.5.1.1 Observation-Based Studies

Observations in Industry

Bryant et al. (2006) used 23 hours of professional pair programmers' dialogue as a source for the pairs' collaboration. The researchers found that the pairs had a high number of verbal interactions: more than 250 per hour, with tasks such as refactoring and writing new code indicating the highest amount of collaboration, and tasks such as commenting indicating the lowest amount of collaboration. For the purposes of this

study, social interactions that were not related to the tasks at hand were not analysed by the researchers.

Chong and Hurlbutt (2007) present data from a four-month ethnographic study of professional pairs from two development teams, visiting each team on a weekly basis. During each observation period, the researcher sat behind one pair "taking notes on their interactions and activities", recording and transcribing their dialogue to produce a detailed record of each session, and identify repeated patterns of behaviour. Similarly, Fronza et al. (2009) observe experienced developers working "in real environments" for ten months, to understand the effects of pair programming on the introduction of new members into the team. Whilst this paper presents certain statistical methods used for data analysis, the authors do not explicitly mention how their observations were carried out.

Observations in Academic Contexts

Murphy et al. (2010) observed several pairs of undergraduate students whilst they debugged a pre-defined set of Java programs with logical errors. The pairs' verbal interactions are transcribed, and each pair's success rate is compared to the number of transactive statements exhibited in their communication to identify any correlations. In the paper, a transactive statement is defined as "a conversational mode in which participants respond to their partner's statements to clarify their own understanding". During the first stage of the study, the authors provided student pairs with 16 programs with logical errors, and asked them to solve as many of them as they could within a 45-minute time limit, recording video and audio for later analysis (an approach used in Chapter 6 of the present thesis). Following this stage, verbal interactions were

transcribed, and all utterances that were related to the students' reasoning were categorised. Table 4 shows the categories used to partition the Murphy et al. data:

Table 4: Analysis of discussions within a debugging context

Category	**Description**
Feedback Request	Do you understand or agree with my position?
Paraphrase	I can understand and paraphrase your position or reasoning.
Justification Request	Why do you say that?
Juxtaposition	Your position is X, and my position is Y.
Completion	I can complete or continue your unfinished reasoning.
Clarification	No, what I am trying to say is the following.
Extension	Here is a further thought or elaboration.
Critique	Your reasoning misses an important distinction, or involves a questionable assumption.
Integration	We can combine our positions into a common view.

The authors found that conversations between pairs included statements which were not necessarily about transactive reasoning or about problem solving in particular, but which still led the pair to an eventual solution. This led to the authors being unable to categorise certain aspects of the observed communication that could not be described by one of the categories in the table shown above. For example, within one of the transcripts, a pair is discussing an accident involving a broken phone. Whilst not transactive, the authors report that "this sort of chitchat […] may help partners become more comfortable working with each other". The authors indicate that over half of the potential enumerated transactive statements were rarely detected in the coded transcripts: "Justification occurred infrequently, and paraphrasing, clarification, and integration were used rarely. Juxtaposition was not detected at all" (Murphy et al.,

2010). Due to the small sample size (5 pairs) and the issues discussed above, no definitive conclusions were drawn from the study but it nonetheless hints at the importance of non-transactive (such as off-topic conversations) statements.

Sfetsos et al. (2006) observe communication within student pairs by counting communication events, which were then divided into the following topics: requirements gathering, specification and design changes, code, unit tests and peer reviewing. This approach was later used by Stapel et al. (2010), where a similar classification was employed and expanded upon to yield the metrics shown in Table 5. These metrics were then used to measure communication within a pair programming situation:

Table 5: Measures of communication in pair programming

ID	**Name**
M1.1	Number of conversations about requirements
M1.2	Number of conversations about design
M1.3	Number of conversations about code
M1.4	Number of off-topic conversations
M1.t	Time of conversations
M2.1	Proportion of drivers' share in conversation
M2.2	Number of questions per hour
M2.3	Number of navigators' clarification quest.

The metrics generated by Stapel et al. (2010) were gathered by one of several external observers watching each pair session and filling in a data collection sheet for each pair. This approach allowed for the intra-pair communication to be quickly segmented in real-time. However, as the pair conversations were not recorded, the authors focused only on counting instances of communication, rather than delving more deeply into

exploring its content or surrounding context. The authors therefore do not investigate how the communication between the programmers is structured (i.e. the pragmatics as defined by Morris (1939)), but with the amount and the content of the conversations as related to a specific programming situation (i.e. requirements, design, and code).

The use of multiple external observers can lead to data being collected in an inconsistent manner. For example, Stapel et al. (2010) report that the number of conversations per hour decreased over each pair programming iteration during their study. However, the authors note that some of the observers used could have missed certain utterances made by the pair.

The literature discussed thus far mostly consists of studies that use an observer to gather communication data. Bryant et al. (2006) indicate that whilst verbalisation occurs naturally in pair programming, future researchers should consider the effects that the observer has on this verbalisation. Within the studies discussed so far, there is no mention of completely removing the observer from the study in order to maintain an environment that is as natural as possible.

Several existing studies use coding schemes to analyse different aspects of pair programming: these will be listed in the following section, alongside a discussion to determine the need for the creation of a new coding scheme.

2.5.1.2 Coding Schemes presented within Existing Studies

Coding Schemes in Industry

Bryant et al. (2006) describe the creation and analysis of a coding scheme with 12 items, designed to analyse a pair's collaboration on certain sub-tasks. The generated items are shown in Table 6.

This coding scheme focuses mostly on the pair's code writing process, and is derived from an analysis of the pair's verbal interactions. Some items that are relevant to communication are included in this coding scheme; for example, *correspond with 3rd party* and *discuss the IDE*, but the authors comment that "instances of social chat either within or outside the pair were not considered".

Table 6: Coding scheme describing generic 'sub-tasks'

Code	**Explanation**
Agree strategy/conventions	Including approach to take, coding standards and naming conventions
Configure environment	Setting up paths, directories, loading software, etc.
Test	Writing, running, or assessing the success of tests
Comment code	Writing or modifying comments in the code
Correspond with 3rd party	Extra-pair communication: person to person, telephone
Build, compile, check in/out	Compiling and building on own or integration machine
Comprehend	Understanding the problem or existing code
Refactor	Re-organising the code
Write new code	Creating completely new code to complete the task
Debug	Diagnosing, hypothesizing and fixing bugs
Find/Check example	Looking at examples in books, existing code or on-line
Discuss the IDE	Talking about the development environment

In 2007, Freudenberg et al. discuss the creation of a coding scheme to analyse the level of detail of the pair's statements. This coding scheme was largely derived from an existing, more general one created for programmers by Pennington (1987), and consists of five codes: *Syntax*, *Detailed*, *Program Blocks*, *Statement Bridges*, and *Real World* (Freudenberg et al., 2007). This coding scheme focuses on the pair's verbal communication, with a particular emphasis on the pair's relationship with the code that is being written.

Plonka et al. (2011) present a coding scheme to investigate more intricate aspects of pair programming; their codes focus on exploring the roles each member of the pair had in initiating role switches, as well as analysing the time each developer spent in the role of pair 'driver'.

Coding Schemes in Academic Contexts

Salinger et al. (2008) and Salinger and Prechelt (2008) discuss the development of a coding scheme created directly from pair observations. This coding scheme consists of over 50 different codes derived from the analysis of one student pair. These codes break the analysed communication down to a particularly fine grain, and allow the authors to generate a general-purpose coding scheme of pair programming activities.

2.5.2 **Adopted Research Methodology**

Quantitative methods focus on the development of metrics and the testing of hypotheses through the collection and analysis of numerical data. Measurement is central to this type of research, which is described as having an "objectivist" conception of social reality. On the other hand, qualitative research "is a situated activity that locates the observer in the world" (Denzin and Lincoln, 2005), thus involving an approach which

allows for subjective interpretation and the emphasis of *words* in data analysis, with a focus on the generation of theories (Bryman, 2012).

A review of the literature demonstrates a preference for qualitative research when analysing communication within pair programming. This is not surprising, since qualitative methods are particularly well suited to understanding complex, "messy", naturally occurring phenomena. As such, they can be usefully applied to understanding the communication that is exhibited by pairs, in a natural setting (e.g. the workplace). A qualitative research method allows for in-depth observation and analysis of the participants in these settings.

2.5.2.1 Grounded Theory

Grounded theory is a systematic methodology that has become one of the most widely-used frameworks for analysing qualitative data (Bryman, 2012). It involves an analysis of data through observations, interactions and materials gathered by researchers; giving guidelines about how the research should proceed (e.g. how to identify categories and how to establish relationships between them). Furthermore, this method can complement other approaches to qualitative data analysis such as ethnography, rather than stand in opposition to them (Charmaz, 2006). According to Myers (2008), grounded theory is "very useful in developing […] descriptions and explanations of organizational phenomena".

The original defining components of grounded theory include the following (Glaser and Strauss, 1967, Dick and Zarnett, 2002):

- Simultaneous involvement in both data collection and analysis;

- Constructing analytic codes and categories from data, not from preconceived hypotheses which have been logically deduced;
- Constantly comparing the data and resulting analysis during each stage of the process;
- Advancing theory development during each step of data collection and analysis;
- Memo-writing to elaborate categories, specify their properties, define relationships between categories, and identify gaps;
- Sampling aimed toward theory construction, not for population representatives.

Grounded theory allows data coding to be used in order to categorise complex behaviour, as well as to promote constant refinement of the data gathered through four key stages (Charmaz, 2006, Glaser and Strauss, 1967):

1. Open coding

   This first stage seeks to gather participants' views, feelings, intentions and actions, as well as the contexts and structures of their behaviour. This stage focuses on writing extensive field notes of observations, and/or compiling detailed narratives.

2. Construction of analytic codes from the data

   In this stage, the initial observations are grouped into related concepts (Lazar et al., 2009), which allow the researcher's analytic grasp of the data to begin to take form.

3. Continual comparison of the data with the codes

   In this stage, the concepts and the categorised data are iteratively refined in order to allow the researcher to create an increasingly detailed and internally consistent interpretation for each concept reflected in the evolving analytic codes.

4. Formation of a theory

   In this final stage, the researcher creates inferential and predictive statements about the phenomena that have emerged in the preceding stages. Connections and correlations are posited and tested, linking the multiple concepts and categories identified in previous stages (Lazar et al., 2009).

Analytic coding is a key process in grounded theory, where segments of data are assigned to categories with descriptive labels, or keywords. The data is coded as it is collected. Unlike quantitative research which requires numerical aggregation and testing, when using a grounded theory approach, the researcher's interpretations of the data shape his or her emergent codes (Bryman, 2012).

According to Charmaz (2006), during initial coding, the following questions should be asked in order to allow for the data to be distilled and sorted, making it easier to compare segments of data with each other:

- "What is this data an instance of?"
- "What does the data suggest?"
- "What theoretical category does this specific segment indicate?"

When constructing codes, it is suggested (Glaser, 1978, Attride-Stirling, 2001) that 10 to 15 analytic codes are typically enough: having too many codes can lead to the possibility of *over-coding* the data. This causes the impact of the core variables to be diluted, as usually only a subset of the codes occur with enough regularity to play a substantial role in explaining overall behaviour (Glaser, 1978, Corbin and Strauss, 2007, Sheridan and Storch, 2009).

The use of grounded theory has a number of advantages over other qualitative research methods: i) it provides a systematic approach to analysing text-based data, allowing researchers to use coding to generate evidence, whereas other qualitative methods often rely upon the application of general principles rather than a systematic method; ii) it is an intuitive process for novice researchers and analysts which gathers rich data from the experience of individuals. Due to the early data analysis stage, a grounded theory approach encourages constant refinement of the theory through frequent comparisons between data collection and the data analysis (Myers, 2008, Lazar et al., 2009).

One of the main disadvantages of this research method is that novices are at risk of finding themselves overwhelmed by data whilst doing the detailed and thorough coding that is required of them. This can make it difficult for a novice researcher to identify the higher-level concepts and themes during their data analysis. Furthermore, the coding and transcribing stages are highly time-consuming, and depend largely on the familiarity of the researcher with the topic at hand, which could be subject to bias (Lazar et al., 2009, Bryman, 2012).

2.5.2.2 Grounded Theory in Software Engineering

Grounded theory is gaining popularity as a way to study the human aspects of software engineering (Adolph et al., 2011). Researchers have used a grounded theory approach to study the self-organising nature of agile teams (Hoda et al., 2010), to understand how software process improvement is applied in actual practice (Coleman and O'Connor, 2008), to explore how people describe software processes in natural language (Crabtree et al., 2009), to understand and analyse behaviour that participants have exhibited whilst asking questions to each other (Sillito et al., 2006), and to outline customer-focused practices in XP teams (Martin et al., 2009). Using grounded theory as a research method allows for the extraction of rich data from the experience of the participants.

Grounded theory has also been used to observe and understand various aspects of pair programming. However, a number of authors adapt the recommended approach, preferring to use predefined coding schemes, and do not tend to use raw observations directly in the analysis process (Salinger et al., 2008). For example, Bryant (2004) uses grounded theory, but uses pre-defined categories, and does not allow for these to evolve and change throughout the observation. Later studies by the author (Freudenberg et al., 2007, Bryant et al., 2006) describes the use of transcribed audio files, which are then coded according to a list of pre-defined set of categories. This is used to generate data which allows for the investigation of behaviours necessary to discuss the debated hypothesis.

Researchers have also used grounded theory approaches in pair programming contexts in order to extract and classify information based on interviews. For example, Ho et al. (2004) describe a study where grounded theory was used to code textual data of pair

interviews in order to structure the data and define categories. These categories were subsequently used as a classification method in order to extract common themes (such as enjoyment, study habit, and pair effectiveness) emerging from the discussed topics. Similarly, Kinnunen and Simon (2010, 2011) use grounded theory to analyse students' interviews, which allows them to develop an analysis of how students perceive their self-efficacy during programming tasks. More recently, Jones and Fleming (2013) use methods from grounded theory, such as open coding, to analyse observations of 14 students involved in a pair programming task. The authors mention that this approach helped them to identify and code various concepts which were becoming apparent in the video data.

Salinger et al. (2008) and Salinger and Prechelt (2008) report the use of grounded theory in order to derive a coding scheme for the objective description of pair programming sessions independent of a particular research goal. The methodology differs from other reported studies that use grounded theory, in that the authors work directly on the raw video data rather than using transcribed data. This is due to the fact that "too much relevant information in the screen recording" made working from transcripts seem impractical.

This exploration shows that whilst grounded theory has been used in software engineering and pair programming contexts, researchers tend to use pre-defined coding schemes and base the coding process on transcribed data in the first instance, rather than using the raw data yielded by observations (e.g. video files) to help direct the way the codes and categories change.

2.5.2.3 An Approach Inspired by Grounded Theory

A framework for the coding and analysis of verbal data has been described by Chi (1997) and was subsequently used in the context of pair programming by Bryant (2004) for the observation of industry-based pairs. This process consists of the following stages:

1. Developing a coding scheme through open coding;
2. Segmenting and coding the sampled transcripts based on the coding scheme;
3. Seeking and interpreting patterns.

This framework can be applied to the grounded theory approach in order to create stages upon which the observation stages of this thesis can be built: communication data can be collected (via ethnographic observations, videos, etc…) and then used for data analysis purposes. By being immersed in the data, the researcher can develop codes and categories, which can continue iteratively until no new categories or properties emerge from the gathering or analysis of further data (Montgomery and Bailey, 2007, Glaser and Strauss, 1967). The analysis of these codes can then lead to patterns of pair communication being drawn out from the data.

2.5.3 **Alternative Methodologies**

An approach inspired by grounded theory methods is considered to be flexible enough to allow for an in-depth analysis of communication data. The following section discusses alternative qualitative methodologies that are frequently used for qualitative studies (Ritchie and Lewis, 2003, Wertz et al., 2011, Bryman, 2012). These

methodologies have been reviewed in order to determine their suitability for the observation of pair communication, alongside grounded theory.

2.5.3.1 An Overview of Qualitative Research Methods

a. Ethnography/Participant Observation

Ethnographic studies are concerned with understanding the social world of people who are being studied through the researcher's immersion and subsequent engagement within their community for a length of time. The results produced are largely descriptive, and detail the way of life of particular individuals, groups or organisations regarding their culture and their beliefs (Ritchie and Lewis, 2003). The researcher is concerned with observing behaviour, listening to conversations, and providing a detailed account of what was observed.

Ethnographers can assume four roles, allowing them to be either completely immersed or completely detached from the observed group (Gold, 1957, Bryman, 2012):

1. Complete participant

   The researcher who is a complete participant assumes a completely covert role, in that their true identity is not known to the members of the group being observed. Thus, the researcher becomes a fully-functioning member of this group.

2. Participant-as-observer

   While the researcher is still fully involved with the group being observed, members of the group are aware of the researcher's role within their community.

3. Observer-as-participant

   The researcher undertakes minimal participation within the group, detaching himself or herself to the point where they are mainly an observer. This occurs in situations where being genuinely involved within the group is difficult; e.g. due to legality or professional issues.

4. Complete observer

   The researcher refrains completely from interacting with the group, instead using methods of observation that are unobtrusive. According to Bryman (2012) most authors take the view that, due to the very nature of this role, researchers acting as total observers are not undertaking ethnographic research since they are not actively participating within the group.

Having a completely external observer can lead to tarnishing the 'natural' setting being observed or affecting the way the participants behave. Stapel et al. (2010) suggest that the observer should be a member of the team of pair programmers (e.g. a student from the paired class), with the role of observer switching to a different team member for every session. This approach, while eliminating an external observer, introduces another complexity: the data is now gathered by a different person in each session. This data is not just prone to human error, but also to classification errors: an utterance might count as a "short conversation" to one observer, but as something that can be discarded to the next one.

In these scenarios, it is also possible that the role of the observer could be completely eliminated by using recording devices (such as an audio recorder placed on the desk, or screen/audio capture). This would allow a researcher to gather communication data during the pair task, which can then be analysed. Whilst not completely observer-free,

this approach would solve the issues discussed: primarily, that of having an external observer influencing the "natural" flow of conversation, and secondly, that of having different observers collecting different data. Recording audio or video also allows the researcher to refer back to the data at any point during the analysis stage should this be required, rather than relying on observation field notes. This allows for a deeper analysis to be undertaken.

b. Phenomenology

The phenomenological approach is a descriptive study of human experience. It questions how individuals make sense of the world, and uses methods that attempt to see things from the participant's point of view. Wertz et al. (2011) describe two fundamental procedures that are necessary for the study of experience:

- Putting aside natural and scientific knowledge about the subject being investigated, and
- The focus on the existence of objects independent of the experience being investigated, as follows:

  "In studying experiences of automobile accidents, the phenomenologist focuses on the way drivers attribute fault to themselves and to others, including all the meanings and consequences of fault as experienced by drivers, without investigating or judging the objective existence of fault, which is the focus of judges and insurance adjusters. The phenomenological attitude is reflective (Wertz et al., 2011)."

Phenomenology is a useful orientation when the researcher wants to be immersed in the meaning of events such as conversations and texts (Ritchie and Lewis, 2003), as it aims

for a collective analysis of individual experiences. The approach requires a deep immersion into the experience that is being investigated and is not appropriate for constructing theories or for testing hypotheses.

A key rule of phenomenology is to put aside *a priori* knowledge about the topics being investigated. However, interpreting communication when observing pair programmers requires a certain level of knowledge about programming and about pair functionality (e.g. when listening to discussions about unit tests, methods, or driving the code). It is expected that existing knowledge of pair programming needs to be used in order to identify key codes and themes within the existing communication exhibited by the participants. Due to this, a purely phenomenological approach is not ideally suited to the understanding of pair communication.

c. Narrative Research

Narrative research is an approach that connects people's lives as if they were a plot, consisting of beginning, middle and end points. It is an interpretative methodology that aims to understand how people interpret and react to their environment, taking into account the connections in people's accounts of past, present and future events and affairs (Bryman, 2012).

The aim of narrative research is not to generalise, or to reveal an underlying 'truth', as it generally takes place with small samples that are not deemed to be representative. This method offers the possibility of exploring aspects of experience that can be used in a pilot study to generate potential research questions (Wertz et al., 2011) but is not aimed at providing generalisations across a larger population.

Summary of Alternative Methodologies

This research aims to present a better understanding of pair communication, in order to lead to an increased knowledge of implications that this communication has on the pair's effectiveness.

Phenomenology requires the researcher to put aside all previous knowledge of the topic at hand – however, it is expected that the interpretation of the pair's communication will require, on some level, an understanding of the topics discussed. Narrative research does not aim to provide generalisations – but it is expected that any information gained throughout this research should not be constrained to the sample population, but aim to be generalised beyond this setting.

Alternatively, an ethnographic approach is still viable at this stage, and may be used to inform the planning of observation sessions with the various pairs.

2.6 Summary of the Literature

Through an examination of existing literature, it can be seen that pair programming has benefits and issues. Communication is seen as being an integral contributor to the success of a pair programming session, but invariably, it is also one of the main causes of failure, and is therefore seen as one of the greatest barriers to pair programming. Furthermore, it can be seen that novice pairs are less able to communicate opinions and ideas within their pair, when compared to more experienced pairs.

This review has identified a gap in the literature: despite a significant amount of research into pair programming, it is not fully clear how various sorts of verbal communication acts within a pair contribute to success. It has been seen that guidelines and instructional materials have been developed in order to assist instructors to

introduce pair programming, and to assist solo developers to get used to start working in pairs – however, it is not clear how these guidelines could be used to assist in the development of useful communication skills.

A better understanding of communication within pair programming could lead to improved teaching practices for pair programming novices, which in turn would allow them to communicate more effectively within their pairs.

The review of various qualitative analysis methodologies and predefined coding schemes suggests that an approach informed by grounded theory will be worthwhile and that this approach opens up the possibility of a different coding scheme being extracted. It is expected that fresh data collection and analysis will lead to a much better understanding of how pair communication is structured, and how the pair's work is influenced by this communication.

2.6.1 Defining *Novices* and *Experts*

Dreyfus and Dreyfus (1986) identify a *novice* as someone with little situational perception, who rigidly adheres to taught rules, and applies no discretionary judgement. At the other end of the spectrum, an *expert* no longer relies on taught rules, has an intuitive grasp of situations based on deep understanding and has a clear vision of what is possible.

In their work on understanding communication structure in pair programming, Stapel et al. (2010) define novice pairs as "new to pair programming and unfamiliar with each other". The novices are compared and contrasted with professional pairs – or "experienced" pairs from industry.

In the context of this thesis, the term *novice* applies to student participants who have little to no previous experience of pair programming. Studies with students who are novice to pair programming are reported in Chapters 5 and 6 of this thesis.

Winslow (1996) suggests that the transition from novice to expert is expected to last about ten years. Although an expert is considered to be someone who has had considerable experience in pair programming. Within the context of this thesis, the term *expert* applies to industry-based pairs. All pairs recruited as experts were required to have had at least six months of full-time commercial pair programming experience. Observations with industry-based pairs are reported in Chapters 3 and 4 of this thesis.

2.7 Research Question

The present research will begin by observing experienced industry-based pairs and analysing their communication. The methodology for this research is based on the particular grounded theory framework for the analysis of communication set forth by Chi (1997), discussed above. It is expected that by limiting the initial observations and investigation to experienced industry-based pairs, an understanding of how these pairs communicate can be developed.

The next stage in the research will then be to apply the knowledge gained from this investigation to novice student pairs. Williams et al. (2008) outline several guidelines for implementing pair programming in an educational setting, based on their experience with over 1000 students (Hanks et al., 2011). In their paper, the authors contend that students need to be trained in order to successfully pair: “the instructor cannot assume that the students will know what to do if they are [simply] told to pair program”. Research shows that whilst pair programming is beneficial in industry and in academic settings (Williams and Kessler, 2002, Begel and Nagappan, 2008, Hanks et al., 2011),

'communication' is still seen as one of the main issues surrounding pair programming by pair programming newcomers in both contexts (Williams et al., 2000, Sanders, 2002, Begel and Nagappan, 2008). It is therefore important to explicitly train these newcomers on how to communicate effectively while pairing.

In the context of this thesis, knowledge about how experienced pairs communicate is derived from industry-based pairs ("experts"), and applied to the training of first-time student pairs ("novices"). It is expected that this applied knowledge may help these novice pairs improve the way they communicate - thus answering the following question explored in this thesis:

> **Can extracted communication patterns from expert pair programmers be used to help novice student pairs to improve their intra-pair communication?**

Chapter 3: Informative Study

Observing experienced pair programmers working together will provide information to help understand how they communicate with each other. This chapter discusses initial observations carried out on videos recorded by one industry-based pair, following an iterative grounded theory approach. A coding scheme is created based on the observed topics and trends of the verbal communication being exhibited by this pair. A sample of five videos is then coded and tested, and the coding scheme is further refined. An analysis of the coded videos allows for the development of an understanding of the various observed 'states' of communication, and how the pair transitioned between these different states.

3.1 pairwith.us

A series of videos has been created by two software engineers with the aim of introducing agile software development to a wider audience (Marcano and Palmer, 2009). Both members of the pair are agile coaches and programmers with over ten years of industry experience at the time of filming.

The video output (the *pairwith.us* project) consists of sixty unscripted pair programming videos, all broadcast online between April and July 2009. Following each broadcasting session, the videos were archived without any post-processing or editing. A repository of the videos is made publically available on vimeo.com[2], under the name 'pairwith.us'.

Throughout the *pairwith.us* project, the developers worked on improving upon an existing automated testing tool for software ("FitNesse"[3]). Every time the developers

[2] http://vimeo.com/channels/pairwithus
[3] http://www.fitnesse.org

worked on the project, they recorded and streamed their interactions – thus, the videos are sequential and follow the project throughout its lifecycle.

Each video typically shows the pair's monitor, thus enabling the viewer to see their code as it is being created (Figure 2). An audio feed captures all verbal communication. Later videos also add a webcam feed at the bottom of the screen that shows the two programmers interacting, captured from above their shared monitor. The three streams (audio, video and code) are, for the most part, synchronised.

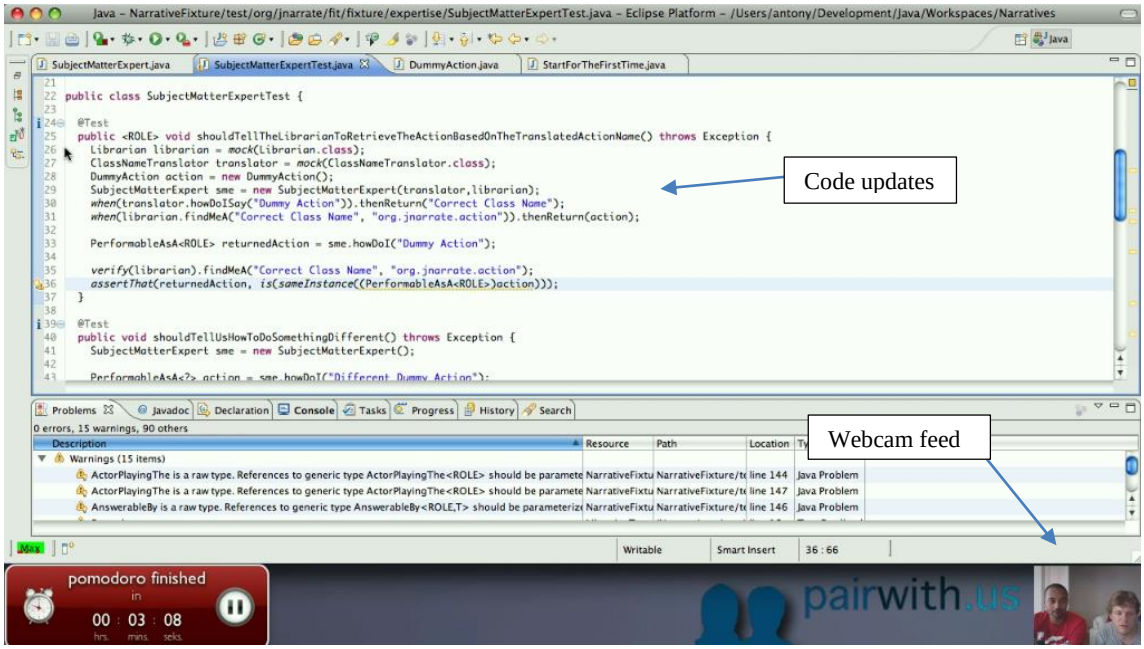


Figure 2: Screenshot of one of the pairwith.us videos

As the pair have more than 6 months' of commercial experience (defined in Chapter 2 as a prerequisite for 'expert' participants), and due to the availability of the current *pairwith.us* repository, this pair and their recorded videos were selected for preliminary observations. The *pairwith.us* team was contacted prior to the study reported here and

gave consent for analysis of their videos, thus enabling initial study of a pair's speech, gestures and actions using qualitative methods (Bryman, 2012).

At the beginning of each session, the pair programmers start a 25-minute countdown timer which will signal the end of their coding session. The video normally ends after the timer has counted down; however, several videos show the pair actively ignoring the timer, choosing instead to pursue their current line of thought. Following the end of the session, the recording is stopped, and the programmers take a short break. Upon their return, they start recording the next video in the sequence using the same setup.

3.2 Creating a Coding Scheme

3.2.1 Methodology

As described in Chapter 2, an approach informed by grounded theory for the analysis of data was adopted. This allows the researcher to be immersed in the extensive data provided by the *pairwith.us* repository, and allows for the iterative development of a coding scheme which would describe the pair's observed communication.

Following an initial viewing of all sixty *pairwith.us* videos, a subset (n=29) was identified that had poor technical quality, such as bad audio-visual quality, lack of video feed, and a prominent, distracting echo (Appendix A). These videos were eliminated from further investigation. The remaining videos that were all recorded in a time-span of three months, between May and July 2009.

A limitation of this study is that it focuses solely on the experiences one pair. There are two benefits nonetheless: (a) initial observations can be drawn from an extensive repository of communication exhibited by an experienced pair; (b) both members of this pair are experienced agile coaches, as well as software developers. The coding scheme

developed from this set of observations is not generalizable, but it does allow for an initial understanding of how a pair experiences various communication states. This coding scheme will be tested against other industry-based pairs in Chapter 4.

Videos provide very rich material that can be a great resource due to the many different kinds of behaviour and context that can be analysed. The researcher has access to not only the participants, but also to their setting, their gestures, their speech, and their activities.

For ease of coding, transcribed data can be easier to search and compare than purely audio and/or visual data. Another advantage of transcription is that the very act helps the researcher identify key themes and become aware of similarities and differences within the data (Bryman, 2012). A full verbatim transcript can be used to highlight features that the analyst deems to be meaningful; e.g. how people speak, and sounds that are not words. Transcribing videos, however, is a highly time-consuming process which can take from two to ten hours or more per hour of video (Chong et al., 2005), and is also subject to human error.

Two approaches were therefore considered at this stage: fully transcribing all the videos, or observing the videos and making annotations. Initially, two of the videos were fully transcribed – a process which took over seven hours. The transcription itself was informative; however, due to the on-going nature of stopping the recording to transcribe it, as well as backtracking through the video to pick up on nuances which were initially missed, it was felt that it would be more productive to be immersed in the data. The balance between transcribing and analysing is referred to by Wetherell et al. (2001): "…it is reasonable and desirable to expect that the extended process of analysis

will identify features which went unnoticed during transcription [...] but it is not feasible to treat transcription as a substitute for thinking and making decisions about the material."

Due to this lengthy process, and the need to be immersed in the data, it was decided to initially view the whole repository of videos whilst annotating interesting communication-related events (a procedure discussed in Chapter 3; specifically, section 3.2). Working directly from recordings (either audio or video) is a good alternative to transcription, "especially for preliminary analysis, such as coding" (Wetherell et al., 2001). This method would allow the researcher to be immersed in the data. It was decided that the best method to adopt for the first stage of the investigation was to work directly from the video recordings, rather than fully transcribing each one. The resulting transcripts were then used to draw out common communication-related topics and themes to inform the development of a preliminary coding scheme. Following this, a sample of five videos was fully transcribed, and these videos were coded using this coding scheme.

3.2.2 **Initial Observations: Open Coding**

Scratch notes (or field notes) are very brief notes that are taken down at the time of the activity, which can be used to jog the researcher's memory at a later date, allowing him or her to recall an account of an event, which can then be expanded on. Scratch notes are considered to be a first step from initial perception to paper (Sanjek, 1990), and are mostly used "in the field", when taking full notes would be too time-consuming and might cause the researcher to miss important data. In the context of this observation, each video was viewed in one sitting with minimal pausing to ensure immersion in the

data: scratch notes were used to capture interesting events and information related to the communication that the pair was exhibiting.

Each video was annotated by hand on a separate sheet as per the example in Figure 3; a typed copy of these scratch notes is given in Appendix B:

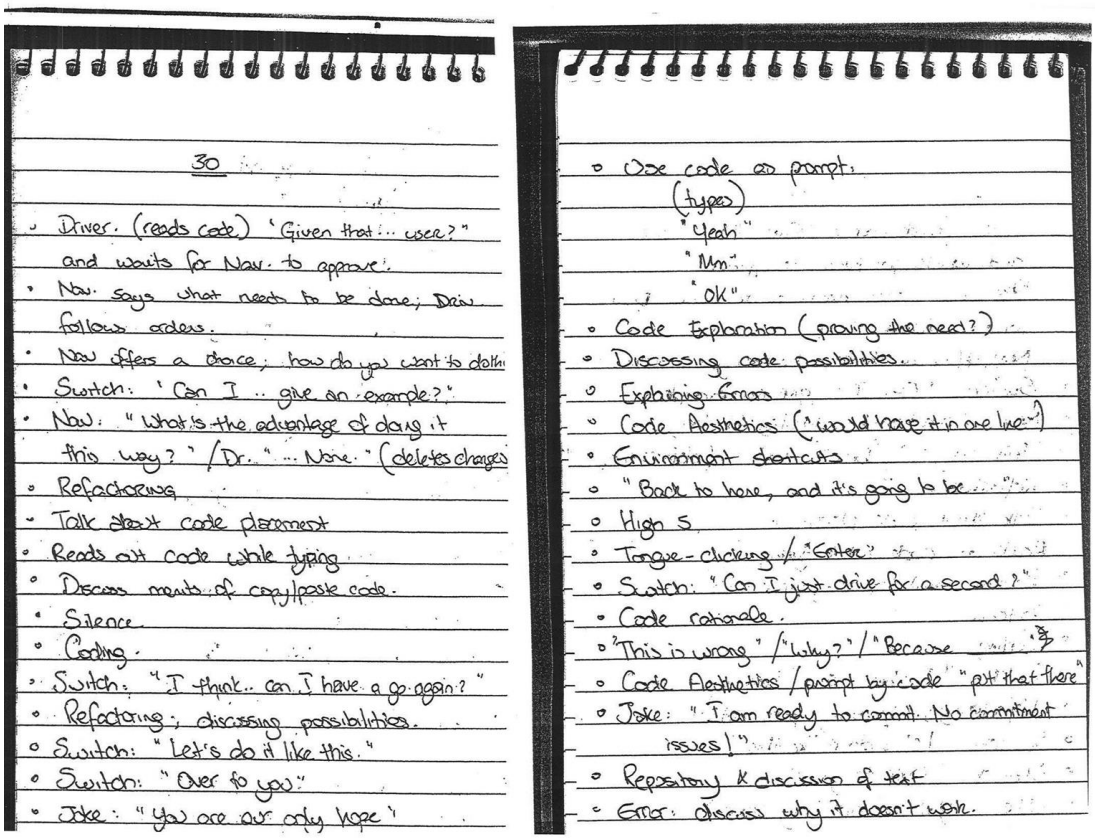


Figure 3: Scratch notes for pairwith.us video #30

As well as the scratch notes, the researcher wrote down a list of observable behaviours, both verbal and non-verbal, noted to be frequently occurring during the pair's communication. These were reviewed and identified during the initial viewing of all 31 videos:

- *Verbal behaviours*
    - Minimal verbalizations used by the navigator to communicate their understanding and feelings to the driver:
        - Communication acts such as deep inhalations of breath, “tutting”, scoffing, and humming while scrolling through code.
        - ‘Mmhmm’, ‘that’s right’, ‘yeah’, ‘OK’, or a simple repetition of what the driver had just said to indicate understanding and acceptance.
        - ‘Don’t know’, ‘hmmm’, ‘no’, ‘actually…’, well…’, ‘except…’ and ‘but’ to indicate various degrees of disapproval.
    - A constant awareness as to why the current task is being carried out, and what is expected to happen following this task, with the following comments seeming prevalent:
        - “Is the code expected to compile, or break?”
        - “What shall be achieved by pursuing these actions?”
        - “How will finishing this task impact the planned next steps?”
    - The driver typically verbalises their programming process by either adopting a ‘think-aloud’ approach, or muttering whilst typing.
    - The driver is seen to ask the navigator for confirmation before proceeding with certain actions.

- The navigator is seen to typically suggest the driver's next steps.
- There is a constant awareness to make the code not only compile, but also look aesthetically pleasing and make logical sense.
- Errors are used as prompting devices for future tasks.
- The switch from driver to navigator is prompted verbally (e.g. the navigator asks to drive).
- Jokes and off-topic conversations seemed to be used more frequently in earlier videos.

- *Non-verbal behaviours*
  - Certain actions, such as pursed lips during moments of uncertainty, or the use of pointing to draw attention to a specific on-screen action.
  - At certain points, both programmers were simultaneously silent. This process was observed to indicate concentration, distraction, uncertainty, or a lengthy programming period.
  - The 25-minute timer is mostly ignored, perhaps due to the fact that the programmers are concerned with getting the task at hand to work, rather than to stick to the time limit.
  - The switch from driver to navigator can be prompted non-verbally (e.g. the driver pushes the keyboard towards the navigator).

At this stage a number of key points emerged that related to elements of the published literature:

1. Bryant et al. (2006) show that within a set of observed expert pairs, the communication distribution between the driver and navigator is 60:40 respectively. This observation could not be verified by watching the *pairwith.us* videos. Typically, identifying the driver and navigator was a straightforward process, but at times the lines between the two roles became blurred. For example, on a number of occasions, both members of the pair would start brainstorming ideas and sketching out possible solutions, effectively acting as two 'navigators'. This mirrors findings reported in studies such as by Chong and Hurlbutt (2007) and Plonka et al. (2011).

2. Stapel et al. (2010) presented metrics (discussed in section 2.5.1 above) including items such as *number of conversations about: requirements; design; code;* and *off-topic*. From analysing the communication within the *pairwith.us* videos, it was clear that it would not be possible to apply these metrics, as conversations did not tend to neatly fit into one of these categories, but generally tended to change from one to the other within the same sentence.

3. The transactive statements used by Murphy et al. (2010) (discussed in section 2.5.1) were compared with the behaviours listed above. Whilst it was clear that some parallels could be drawn (instances of *Justification Request*, *Clarification*, *Completion* and *Extension* were observed), it was also clear that the transactive statements were insufficient to describe all the communication; there were no items that allowed for the categorisation of more social talk, such as off-topic chat and jokes, or items that were not directly related to the process of writing code. This observation was also made by the authors in their paper: "this sort of chitchat [..] may help partners become more comfortable working with each

other." / "We also observed non-transactive discussions that led to solutions" (Murphy et al., 2010).

In general, it is clear that these coding schemes are inadequate to fully characterise and understand communication acts that do not focus directly on discussions related to the task of writing software. Further analyses need to be undertaken in order to develop a more complete understanding of the intra-pair communication that is exhibited by the *pairwith.us* partners.

3.2.3 **Constructing the Coding Scheme**

Following the investigative approach informed by grounded theory outlined in Chapter 2, the data collected needs to be refined into a set of analytic codes. This analysis will allow for a clearer understanding of the communication acts observed during open coding.

The observed behaviours listed above were compared to the video scratch notes (Appendix B) to generate a list of potential analytic codes or keywords that categorise segments of the audio-visual data. This process was carried out by looking for instances of the observed behaviours in the scratch notes and visually annotating these using different colours to represent different communication acts (Figure 4). Each colour group was subsequently named and listed as a separate analytic code. The list of analytic codes thus produced can be used to describe instances of communication within the videos.

Figure 4: Scratch notes annotated with observed communication behaviours

An initial list of analytic codes is presented (in no particular order) below; this is the preliminary coding scheme resulting from the *pairwith.us* observations:

- Talking about previous work
- Continuous planning towards the expected goal
- Silent instance
- Discussion
- Unrelated conversation

- Joke
- Switching of roles
- 'High 5'
- Distraction

3.3 Testing the Coding Scheme

A key stage of grounded theory for the researcher is to continuously compare the data with the generated codes in order to ensure that an internally consistent interpretation can be created for each concept reflected in the evolving coding scheme (Charmaz, 2006). At this stage, the analytic codes needed to be refined and evaluated. To this end, a smaller sample of videos was selected which would be transcribed in a more *verbatim* manner. This reduced sample was coded 'incident-by-incident' using the codes presented above.

Following the evaluation of several possible transcription tools (including NVivo[4], ELAN[5] and ATLAS.ti[6]), Transana[7] was chosen due to its ease of transferring data across multiple machines, as well as due to its simplicity. The software gives certain advantages such as simple keyboard controls that make the transition from watching a video to transcribing it relatively effortless. Furthermore, exporting the coded data (for coding and transcribing on different computers) is easily achieved, allowing for multiple backups during the project lifecycle. The ability to synchronise video playback with the transcript being produced at various points (by using timestamps) was also helpful, and allowed the researcher to view different video segments relating to the

[4] http://www.qsrinternational.com/products_nvivo.aspx
[5] http://tla.mpi.nl/tools/tla-tools/elan/
[6] http://www.atlasti.com/index.html
[7] http://www.transana.org

different analytic codes following the initial coding exercise. This meant that for each analytic code (e.g. Silence) the researcher was able to view all the relevant video clips across all chosen samples to confirm if the codes were correct.

3.3.1 Coding of Sample Videos and Continual Comparison of Data

Five videos were selected at random to represent the full set of 31. Each was imported into Transana and individually transcribed. The transcription process took over a week, again confirming the estimation made earlier on in the process. Each video in the sample was coded using the analytic codes derived from the earlier observations. As per the grounded theory method, this was an iterative process – as each video was coded, the coding scheme was continuously refined and updated to ensure that the coding scheme was consistently valid across the sample videos.

The robustness of the robustness of the coding scheme was tested using an inter-rater reliability after the fifth video to establish if the sample size chosen was sufficient. Using more videos could have strengthened the validity of this analysis; however, the result of 0.718 (see section 3.3.2) indicates substantial agreement across the raters, indicating that that the coding scheme was robust enough to code all instances of verbal communication that were exhibited by the pair.

Each sentence of the conversation transcribed was time-stamped, and linked with the video file for retrieval (Figure 5).

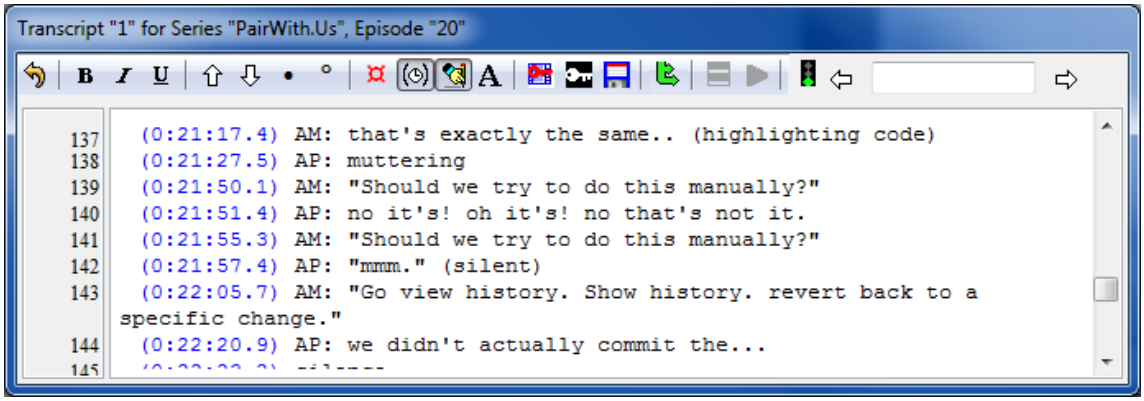


Figure 5: Sample Transcript for Video #53 in Transana

The coding process was initiated once all transcripts were completed. Two approaches of coding were considered: (i) making comparisons between incidents; (ii) making comparisons between lines of the transcript (e.g. where each line of the transcript is assigned to a code). It was found that the former option worked better as this allowed for the researcher to apply codes to every different situation, or incident, rather than being constrained by lines of the transcript. Comparing incidents also allows for the researcher to identify properties of the emerging concept (Charmaz, 2006).

Due to the fact that the sample videos were transcribed in a verbatim manner, the coding process was more thorough than the initial open coding, and allowed for a deeper analysis when applying the codes. This was a two-stage process: initially, video #58 was coded iteratively with the coding scheme above to test robustness, making minor changes to the coding scheme with each iteration. Once the researcher was confident that the changes made had allowed for the video to be fully encoded, the remaining four videos in the sample were coded.

As per the approach informed by grounded theory, this coding process indicated that some of the codes needed to be refined to match with certain situations. Furthermore, for ease of use, some of the codes were renamed to single-word variants. No changes were made to the codes for *Joke*, *Switching of roles*, and *Distraction*.

The following is a list of code changes and adaptations:

- 'Talking about previous work' was renamed to *Review*.
- 'Continuous planning towards the expected goal' was renamed to *Planning*.
- It became clear through the coding process that there were differences between 'pure' silence, and pair muttering (e.g. whilst typing, or figuring out code logic). Thus, 'silent instance' was split into *Silence* and *Muttering*
- The code for *Discussion* was found to be too open-ended and vague. This was split into *Suggestion*, *Explanation*, and *General Talking*.
- 'Unrelated conversation' was renamed to *Off-Topic*.
- *High 5* was seen as a behavioural code (not one which could be matched with a transcribed instance of verbal communication) and was therefore removed from the coding scheme.

3.3.2 **Inter-Rater Reliability**

To counteract the possibility of the coding scheme being tarnished by any preconceptions made by the researcher, its inter-rater reliability (IRR) was analysed. Weber (1990) suggests that the goal of any form of reliability control is to ensure that "different people code the same text the same way".

Colleagues (n=3) from within the School of Computing at the University of Dundee (referred to as 'raters' for the purposes of this section) were recruited in order to perform an assessment of the analytic codes' IRR. All colleagues were recruited from different research groups and independent of this study.

The raters were individually provided with a list of the modified analytic codes (as above), including brief explanations of each code. The researcher gave a brief demonstration of Transana, and asked each rater to code specific samples of all five videos.

Two measures of IRR were computed: Cohen's Kappa (Cohen, 1960) and Fleiss' Kappa (Fleiss, 1971). Both calculate a value of Kappa ranging between 0 and 1.0, with a larger value corresponding to a greater agreement between the raters. A value of 0.61 – 0.80 indicates a substantial agreement, while a value of 0.80 – 1 indicates almost perfect agreement (Landis and Koch, 1977). Cohen's Kappa compares two raters to calculate the Kappa value, whereas Fleiss' calculates an agreement amongst all raters, including the researcher.

The IRR for all raters was found to be Kappa = 0.718 ($p < 0.001$), indicating a substantial agreement.

Individually, the reliability between the researcher and rater A was Kappa = 0.790 ($p < 0.001$), 95% CI (0.674, 0.906). The reliability between the researcher and rater B was Kappa = 0.770 ($p < 0.001$), 95% CI (0.647, 0.893). The reliability between the researcher and rater C was Kappa = 0.729 ($p < 0.001$), 95% CI (0.602, 0.856).

The reliabilities between the raters themselves were also calculated. The reliability between rater A and rater B was Kappa = 0.748 ($p < 0.001$), 95% CI (0.623, 0.873). The

reliability between rater A and rater C was Kappa = 0.665 ($p < 0.001$), 95% CI (0.530, 0.800). The reliability between rater B and rater D was Kappa = 0.606 ($p < 0.001$), 95% CI (0.463, 0.749). These values indicate substantial agreements.

Individual Cohen's Kappa scores are shown below:

Table 7: Cohen's Kappa for the researcher and Rater A

	Value ($p < 0.001$)	Std. Error	% of agreement
Kappa	0.790	0.059	82.46%
Items Coded	57		

Table 8: Cohen's Kappa for the researcher and Rater B

	Value ($p < 0.001$)	Std. Error	% of agreement
Kappa	0.770	0.063	80.702%
Items Coded	57		

Table 9: Cohen's Kappa for the researcher and Rater C

	Value ($p < 0.001$)	Std. Error	% of agreement
Kappa	0.729	0.065	77.193%
Items Coded	57		

Table 10: Cohen's Kappa for Rater A and Rater B

	Value ($p < 0.001$)	Std. Error	% of agreement
Kappa	0.748	0.064	78.947%
Items Coded	57		

Table 11: Cohen's Kappa for Rater A and Rater C

	Value (p < 0.001)	Std. Error	% of agreement
Kappa	0.665	0.069	71.93%
Items Coded	57		

Table 12: Cohen's Kappa for Rater B and Rater C

	Value (p < 0.001)	Std. Error	% of agreement
Kappa	0.606	0.073	66.667%
Items Coded	57		

Following the rating exercise, the raters were interviewed to obtain comments about the coding scheme. Feedback from the raters highlighted the need for further refinements to the coding scheme:

- The raters used *Suggestion* and *Planning* interchangeably, in that they used one to indicate the other on more than one occasion. The raters commented that there was no discernible difference between planning something and suggesting it – thus; the codes were combined into *Suggesting*.

- *General* was deemed to be too broad: any code would theoretically fit under the term. It was renamed to *Code Discussion*, to be used when the pair discuss logic, objects and/or methods.

- *Off-Topic* and *Joke* were combined (as it was becoming more difficult to distinguish an off-topic phrase into its 'off-topic' and 'joke' segments) and renamed to *Unfocusing*.

- *Switch* was judged to be based on the pair's behaviour, rather than their communication, and was hence removed from the coding scheme.

- A *Distraction* was considered to be a feature that was not within the pair's control. Conversations resulting from an external distraction (e.g. a noise outside the office, or a third person interrupting the pair) were considered to be outside the remit of this coding scheme.

3.4 The Coding Scheme

The IRR analysis confirmed that the reliability of the coding scheme was substantial. Nonetheless, comments from the raters suggested several improvements to the codes, as outlined above, that were incorporated for subsequent use. The following coding scheme was used to fully code the sample of five videos from *pairwith.*us. The verbal communication coding scheme thus created consists of the following analytic codes. Each code is accompanied by an exemplar; other examples of how the code was applied in the *pairwith.us* context are presented in Appendix D. In the following conversation transcripts, *N* denotes the navigator, and *D* denotes the driver:

3.4.1 **Review**

The *Review* code was used to describe parts of the session where the pair discussed previous code they had worked on, or legacy code that they were returning to after a period of time. Instances in which the pair reminded each other about pending tasks following the previous session, and instances near the end of a completed task when the pair are marking items off their task-list, were also categorised at *Reviews*.

Figure 6 gives an exemplar of the *Review* code. The conversational fragment starts off by the driver voicing uncertainty about what they did at some point in the past. The pair

review the written code, and re-assert their initial justifications for writing the code and leaving it in its current state.

D: Why - What did we do here?

N: I vaguely recall – I'd – that we were concerned that once you told an Actor to go home, then someone might try to use it.

D: But that would end up with some sort of NullPointerException – a problem, whatever the Actor is acting as a container for…

N: So we just said… we won't let you try and stop that process. We know we've already got rid of it.

Figure 6: Exemplar of a Review

3.4.2 **Suggestion**

The *Suggestion* code was used when the pair was planning the next stages of their work. Typically, a member of the pair would clarify which steps were required in order to achieve a particular goal. This code was also used in instances where the pair discussed possible ways of fixing errors. While *Reviews* looked backwards, *Suggestions* looked forwards, as shown in Figure 7.

N: You need to rename that.

D: OK.

N: And then that should be 'findMeA'… and that should work. (reading errors)

D: We might have to do a plain actorRole… we need to do, um…

N: It's doing the actor thing, so we just need to do the role. We need to return dummyRole.

D: We're just doing the dummyRole.

N: We don't even need to do that.

Figure 7: Exemplar of a Suggestion

It can be seen that the navigator is suggesting that a method needs to be renamed. This is followed by the pair planning actions that are potentially associated with this renaming.

3.4.3 **Explanation**

The *Explanation* code was used in instances where a member of the pair would explain or justify a decision. An *Explanation* was sometimes prompted by the pair trying to understand certain errors, or by a member of the pair explaining the logistics behind legacy code. *Explanations* differ from *Reviews* in that they provide a rationale for the way things are. They differ from *Suggestions* in that they do not propose a future action.

Figure 8 presents the driver explaining in some detail *why* the code is not functioning as expected. In this case, the explanation comes as a reaction to the navigator voicing confusion, or surprise.

N: It's a place to start. We have context.txt and there's the – a – what's happened here?

D: That's the story, isn't it? It's got a new behaviour. The CastingDirector and the actors are here… because the actors have the 'go home' on them. And that looks good for the Therapist class, but it's being created by the… thing.

Figure 8: Exemplar of an Explanation

3.4.4 **Code Discussion**

Code Discussion was used to categorize instances where the pair were making general remarks about the code and the way it was observed to be behaving, or when they were discussing the functionality of their development environment. *Code Discussion* differs from previous codes in that it deals with the context in which the code is being written.

Figure 9 shows the pair discussing various features they were discovering following an unexpected update to the IDE being used (in this case, 'Eclipse').

D: I've got these lazy new shortcuts I never knew I had.

N: What do you mean?

D: I can refactor so much quicker now.

N: Is that new Eclipse? It has so many features. I might move over to it soon. Is it for Linux?

D: No, it's for Mac. The Linux version comes out in a couple of months.

Figure 9: Exemplar of a Code Discussion

3.4.5 **Muttering**

The *Muttering* code was used when a member of the pair was typing at the keyboard or writing down notes on paper, and muttering out loud about what was being written. *Muttering* was seen to occur mostly when the pair was attempting to write or refactor code. *Muttering* differs from the above codes in that only isolated words or incomplete conversational fragments are uttered.

Figure 10 gives an exemplar, showing the navigator working out what needs to be done on a notepad. The driver in this case is busy studying the code on the screen; however, the navigator seems to keep him engaged by muttering out his thoughts.

N: Actors page… if we would do something like that… we have no environments.

Figure 10: Exemplar of Muttering

3.4.6 **Unfocusing**

The *Unfocusing* code identified when the pair were making jokes or taking a break from their programming tasks. *Unfocusing* occurred throughout the session. As examples, the pair would engage in off-topic discussions when faced with a problem that they could not solve. This code differs from all the above in that it was used to categorise instances when the pair was discussing topics that were unrelated to coding or to the current task.

In Figure 11, the pair is choosing to stop their current task and take a break. The discussion in question shows the pair actively choosing to break their focus to pre-empt themselves getting tired.

N: I think we've gone way over the pomodoro.

D: We have gone way over it.

N: Let's – let's just – there's plenty of things reminding us what to do next. I think it's important to maintain our –

D: Rhythm.

N: - Yes, our rhythm. Let's take a break, and then we'll come back. Otherwise we'll get ratty and irritable.

Figure 11: Exemplar of Unfocusing

3.4.7 **Silence**

Throughout the observation of the pair videos, the pair were frequently seen to be sitting together silently. The *Silence* code was used to capture these observed instances. Despite the coding scheme being otherwise focused on verbal communication, this complete lack of verbalisation occurred frequently enough to warrant its inclusion in the coding scheme. Across the sampled videos, 11% of the coded communication was seen to be *Silence* (Figure 15).

When using a grounded theory approach, "the researcher's interpretations of the data" is used to shape the extracted codes (Bryman, 2012). Each instance of Silence in the five sample videos was thus initially identified as per the researcher's developing intuitions. When observing the pairs, a lot of silent intervals (of varying lengths) were identified in their speech; some were quite short, but some intervals (for example, when the pair were trying to work out solutions to a problem) were quite lengthy. For purposes of

clarity and potential replication, these initial intuitions needed to be confirmed via a metric which identifies a cut-off point for distinguishing brief gaps in speech from instances of coded *Silence*.

The shortest observed instance of silence was 1.7 seconds and the longest observed instance of silence was 39 seconds. All instances of the duration of the coded *Silence* in the sample videos were reviewed, and the distribution of silent interval durations was measured (Figure 12), showing a strong skew to the right:

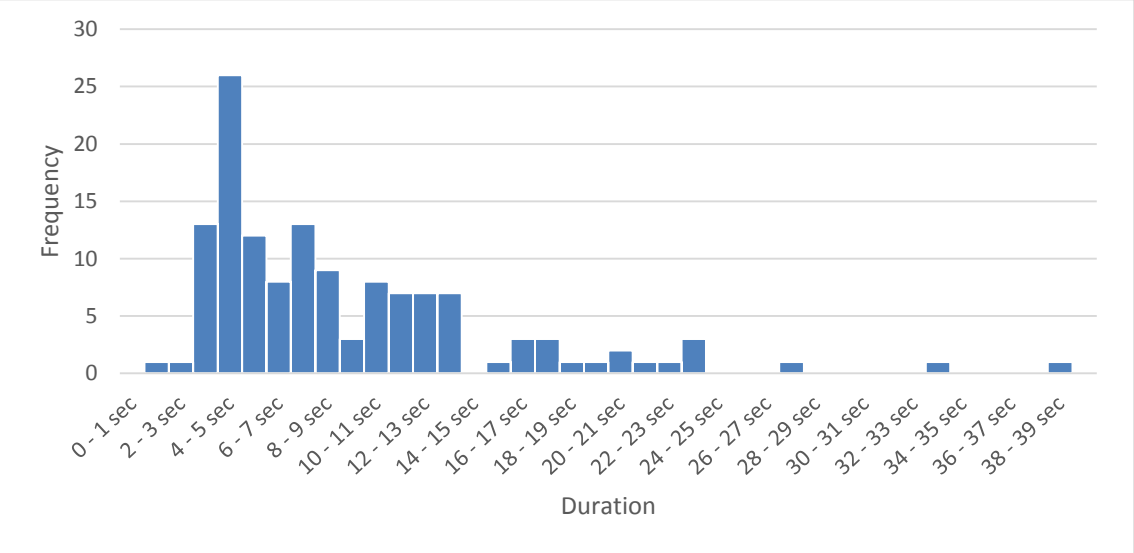


Figure 12: Frequency of durations for Silence.

For the distribution shown above, the mean is calculated to be 9.3 seconds, whereas the median value is 7.6 seconds. A visual assessment of the distribution shows that the median value is closer to the peaks in the distribution (indicating more frequent pauses), whereas the mean value occurs after most of the frequent distributions. Furthermore, Campione and Véronis (2002) present a discussion on silent pause duration in spontaneous speech (reviewed in section 2.2.1.1), demonstrating that when the data is

strongly skewed to the right (as per Figure 12 above), the median value is typically considered to be a more reliable measure of the distribution's central tendency than the arithmetic mean. Thus in this case, the median is considered to be more representative of the central tendency.

For all subsequent analyses, the *Silence* analytic code is therefore used to code each period of silence between the pair that is greater than 7.6 seconds. This does not imply that the eliminated pauses (<7.6 seconds) are unimportant, but merely gives a starting point with which to start analysing and coding the collected data. This cut-off point works well for the observed pair, as it allows for the elimination of the shorter gaps in speech, and leaves the larger 'thinking' silent periods for further analysis.

3.5 Code Analysis

Following the creation of this coding scheme, the next action was to further analyse frequencies or interactions between the codes.

The sample videos used were analysed and coded using Transana. Analytic codes were assigned to each instance of communication, tagging them to the correct location in the video. An example of this coding is given in Figure 13, which shows a 10-minute segment of codes used across the *pairwith.us* sample.

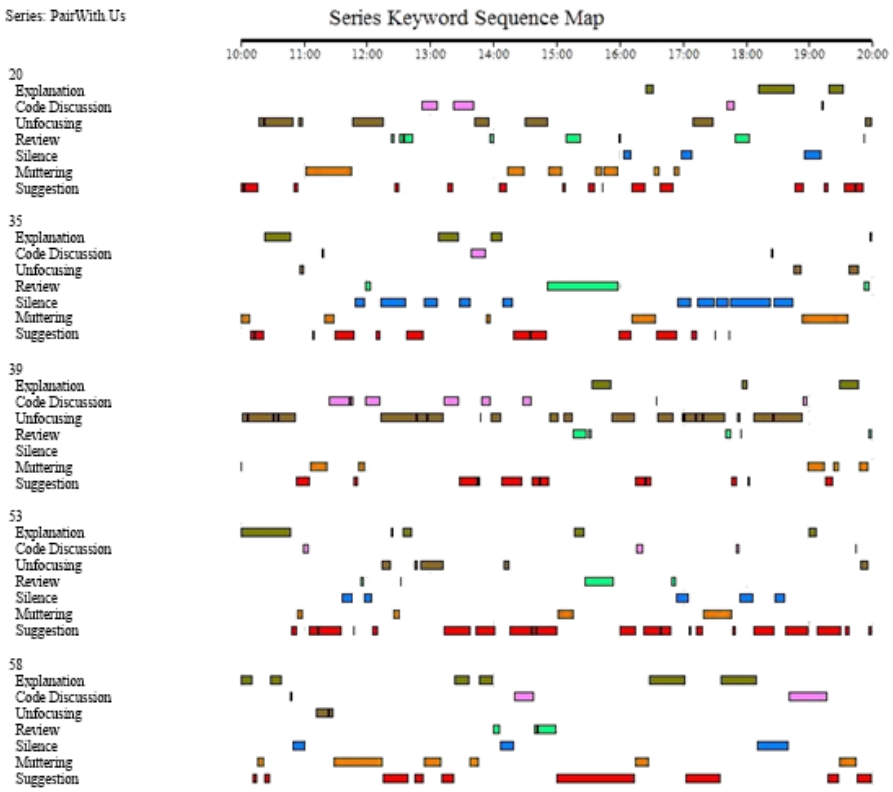


Figure 13: Analytic codes in sample videos (0:10:00 - 0:20:00 mark).

An understanding of how the coding scheme was used across the five *pairwith.us* videos would reveal how the pair's communication was structured. To this end, two aspects of how the codes were used were analysed: the duration of each code and its frequency of use. The following section gives an example of how the coding scheme was used throughout the videos to investigate the structure of the pair's communication.

3.5.1 **Code Duration**

Table 13 presents the total percentage of each video that was successfully coded by the final coding scheme.

Table 13: Percentage of episodes coded from the sample videos

Video Number	Video Duration (min:sec)	Total Time Coded (min:sec)	Percentage Coded
20	28:44.6	27:04.4	94.2%
35	27:29.1	25:41.9	93.5%
39	31:19.2	29:24.7	93.9%
53	25:17.3	23:21.8	92.4%
58	46:35.2	44:31.2	95.6%

On average, the coding scheme is thorough enough to cover 94% of the sampled pair programming videos. This, coupled with the confirmation of the coding scheme's robustness using an IRR (section 3.3.2), was considered to be enough to allow for the coding scheme to be used for subsequent coding and analysis. The following data (Table 14, Table 15 and Figure 14) show the total time attributed to each analytic code per episode, as well as the overall percentage of each episode that was covered by a specific analytic code.

Table 14: Duration of each analytic code

Analytic Code	**Video #20 (min:sec)**	**Video #35 (min:sec)**	**Video #39 (min:sec)**	**Video #53 (min:sec)**	**Video #58 (min:sec)**
Explanation	03:45.7	02:23.8	04:54.1	04:17.5	13:21.1
Code Discussion	01:25.5	00:33.7	02:46.0	00:50.0	03:32.4
Unfocusing	06:03.5	03:15.8	07:01.2	01:59.3	02:46.8
Reviewing	03:19.0	04:22.7	03:42.0	01:32.6	03:51.8
Muttering	03:36.1	03:58.9	03:00.9	03:00.3	03:47.4
Silence	01:37.6	04:53.3	01:06.1	02:40.4	05:19.1
Suggesting	07:17.0	06:13.7	06:54.4	09:01.7	11:52.6
Total Time	27:55.6	27:18.7	30:33.4	24:58.2	44:31.2

Table 15: Duration of each analytic code as a percentage value of the total time coded

Analytic Code	**Video #20**	**Video #35**	**Video #39**	**Video #53**	**Video #58**
Explanation	13.5%	8.8%	16.0%	17.2%	30.0%
Code Discussion	5.1%	2.1%	9.1%	3.3%	8.0%
Unfocusing	21.7%	12.0%	23.0%	8.0%	6.2%
Reviewing	11.9%	16.0%	12.1%	6.2%	8.7%
Muttering	12.9%	14.6%	9.9%	12.0%	8.5%
Silence	6.0%	19.0%	3.8%	11.4%	12.0%
Suggesting	26.1%	22.8%	22.6%	36.2%	26.7%

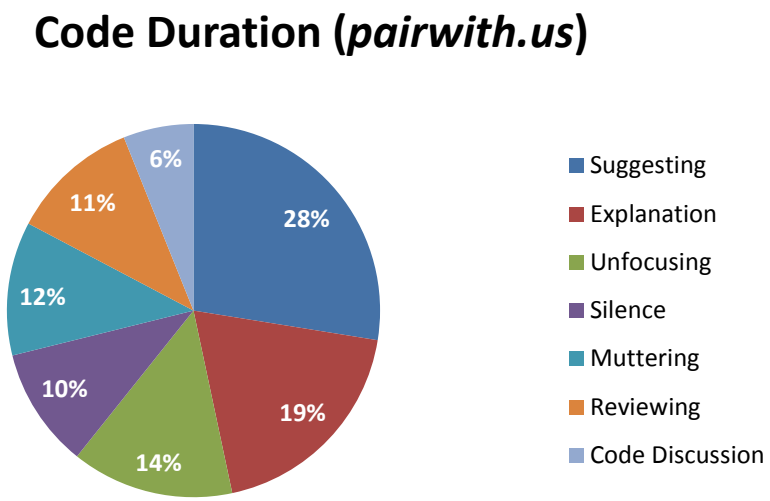


Figure 14: Total duration of codes

The data shows that the *pairwith.us* programmers mostly communicated by making suggestions or explaining things to each other, with these actions taking up 47% of their total communication. The rest of the time was split approximately evenly between unfocusing, on silence, muttering, and on reviewing previous code. Finally, 6% of the total time was spent discussing code logic and placement. It can be seen from the data in Figure 14 above that certain activities (e.g. *Suggesting*) had a longer duration than others (e.g. *Code Discussion*) overall, whereas some activities change markedly per video (e.g. *Silence* and *Muttering*).

3.5.2 **Code Frequency**

Table 16 depicts the total number of times each code was used across the *pairwith.us* samples. The total number of occurrences for each analytic code was counted and the total time 'covered' by each code was calculated, to understand the frequency of use of each code.

Table 16: The total number of occurrences and total time covered for each code

Analytic Code	Number of Occurrences	Total Time Covered (min:sec)
Suggesting	198	41:19.4
Explanation	59	28:42.2
Unfocusing	64	21:06.6
Muttering	75	17:23.6
Reviewing	94	16:48.1
Silence	69	15:36.5
Code Discussion	69	09:07.6

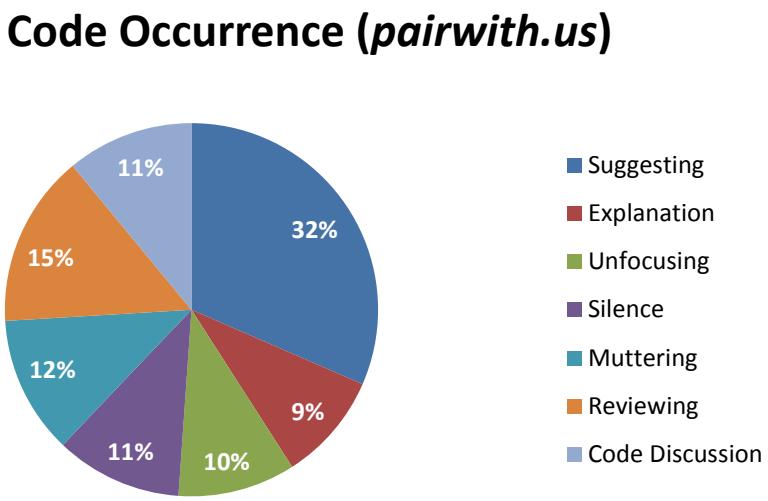


Figure 15: Frequency of code occurrence

The data above (collectively summarised in Figure 16 below) highlights differences between the number of times a code occurs and its duration. As an example, the code for *Explanation* occurs only 59 times, but has greater duration than the code for *Review*, which occurs 94 times. It can be seen that *Suggesting* is a dominant activity, having

been used to code over a quarter of the videos. This is due to the fact that within the pair programming exhibited by the *pairwith.us* team, suggestions were constantly made by both the driver and the navigator in order to drive the work forward.

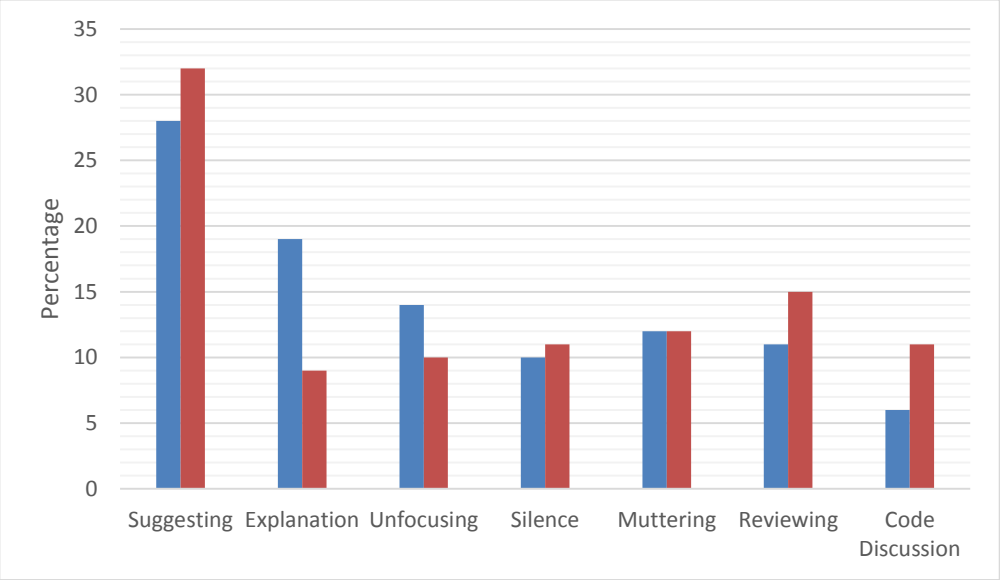


Figure 16: Comparisons between duration (blue) and occurrence (red).

3.6 Pattern Generation

Identifying and validating a set of coded communication states for expert pair programming is the first step towards understanding how expert pairs communicate. It is necessary go beyond this to understand how communication flows from one state to the other (or, more precisely, from one analytic code to the next). In this section, the co-occurrence relationships between the analytic codes will be analysed and discussed.

3.6.1 Transitions between Analytic Codes

Transitions between communication states were examined to identify typical transitions between activities. For this, each code was 'paired' with its subsequent code. For example:

In video #53, the following sequence is coded:

27:01.3 – 27:06.9: Suggesting

27:06.9 – 27:16.3: Silence

27:16.3 – 27:23.5: Suggesting

27:23.5 – 27:38.1: Code Discussion

27:38.1 – 27:56.4: Unfocusing

When looking at how codes progress, the following transitions can be seen in the sequence shown above:

Suggesting → Silence

Silence → Suggesting

Suggesting → Code Discussion

Code Discussion → Unfocusing

This was repeated for all the codes throughout the sample videos.

For each analytic code in the coding scheme (*code A* for the purposes of this explanation), a list was generated consisting of all the codes that could follow it, based on the five coded videos. The occurrence of each code following *code A* was calculated and is given in Figures 17 to 23.

In order to better understand how each analytic code leads to the next, the most common transitions were identified, as follows. The possibility of each occurring transition was calculated to be 1:7 – that is, prior to this analysis, each analytic code has a 1:7 chance

(or 14.29%) of following the current code. This value was taken to be the cut-off point: any occurrences that were greater than 14.29% were considered to be more commonly occurring than the expected value, and thus considered for further analysis. The resulting 'most common' transitions are given in Table 17, with a line indicating the 14.29% threshold.

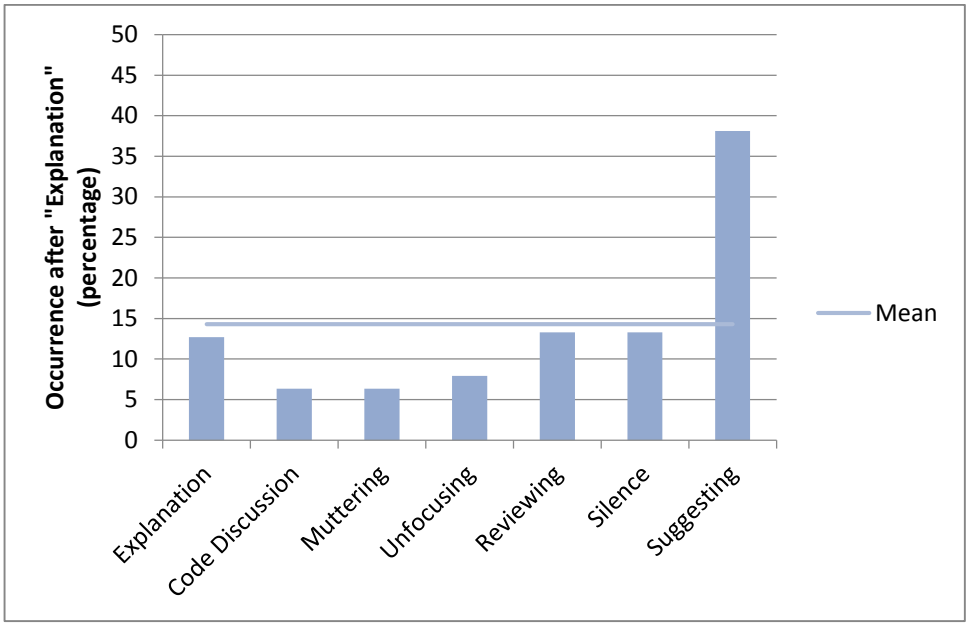


Figure 17: Codes that followed "Explanation"

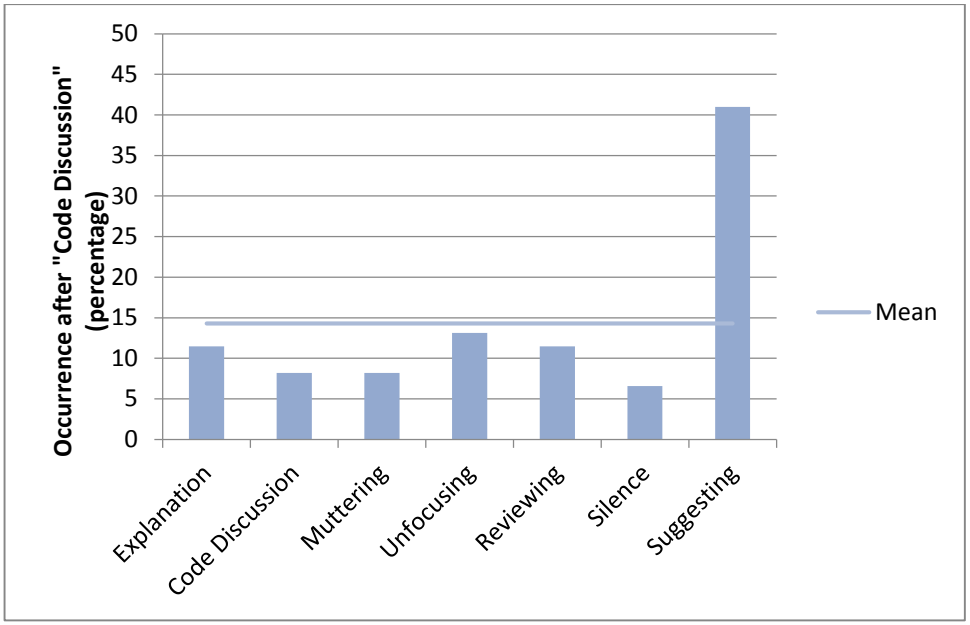


Figure 18: Codes that followed "Code Discussion"

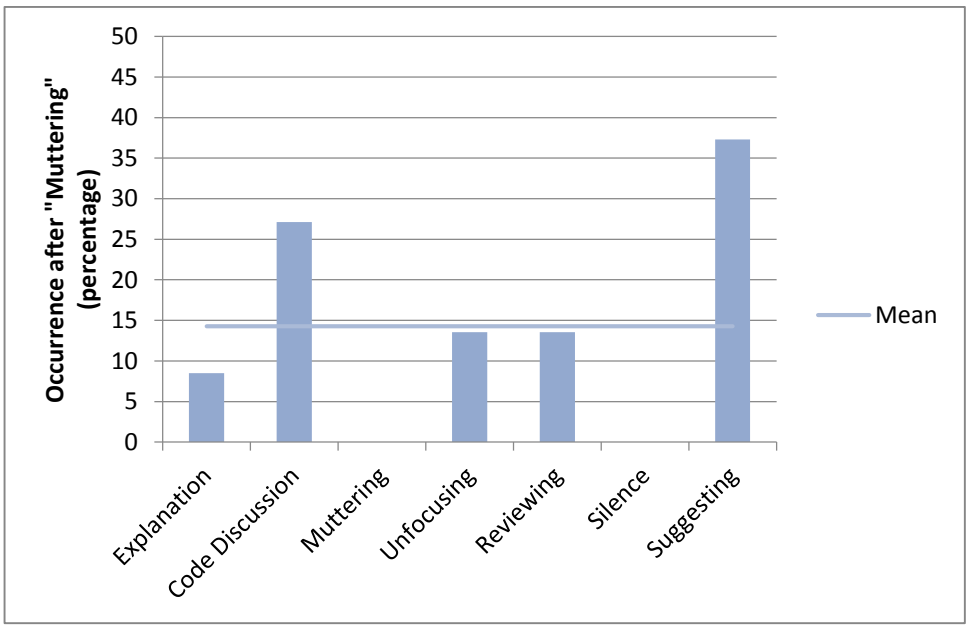


Figure 19: Codes that followed "Muttering"

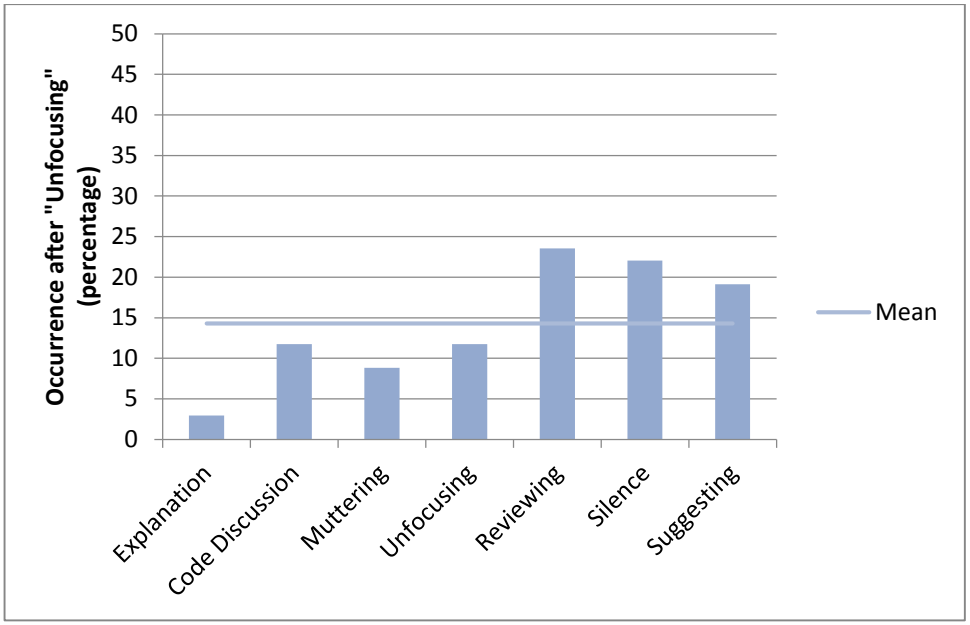


Figure 20: Codes that followed "Unfocusing"

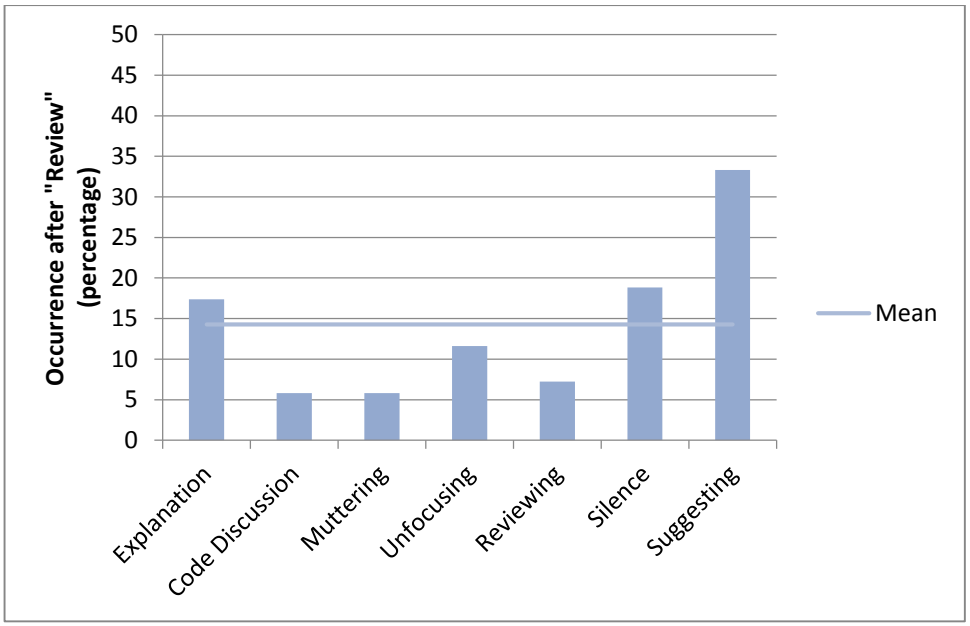


Figure 21: Codes that followed "Review"

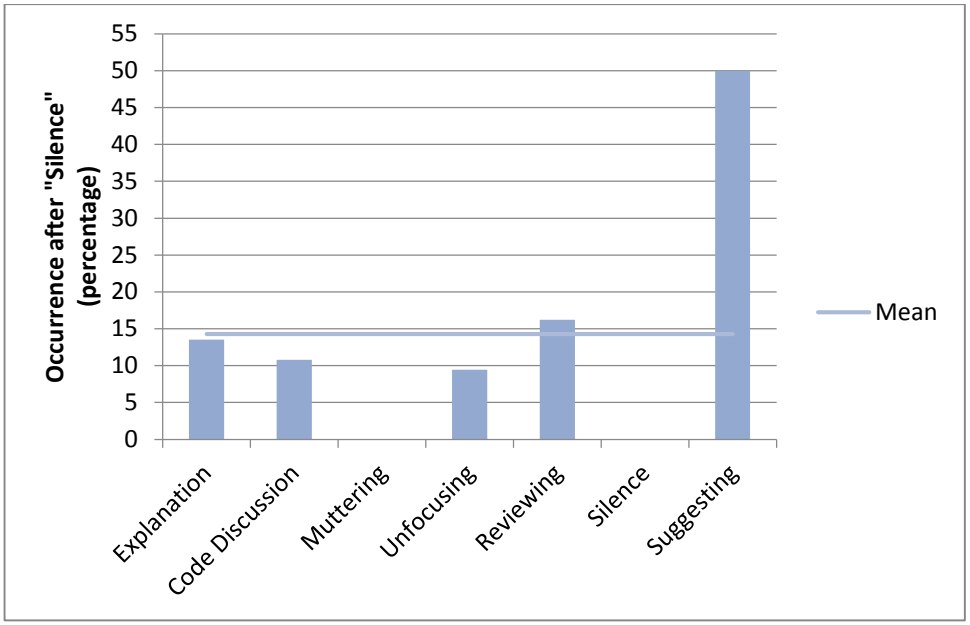


Figure 22: Codes that followed "Silence"

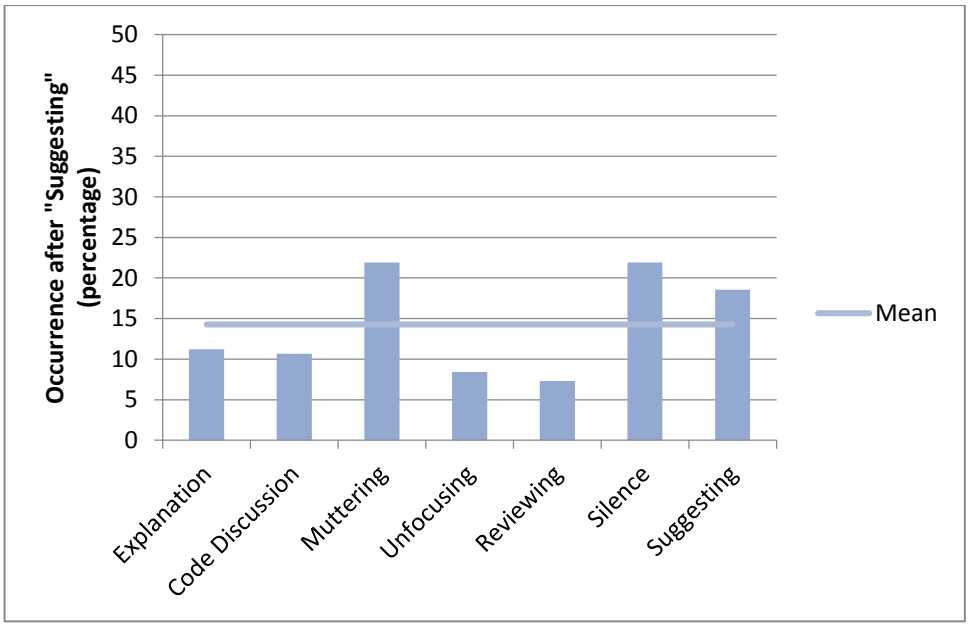


Figure 23: Codes that followed "Suggesting"

The above figures (Figures 17 – 23) show the most common transitions that lead from one code to the next. It is clear, for example, that *Muttering* is followed most commonly

by *Suggesting*, then *Code Discussion*. It is also evident that *Suggesting* occurs frequently throughout the videos, appearing as the most common code following *Silence*, *Review*, *Muttering* and *Code Discussion*.

It can also be seen that in most cases (e.g. Figure 23), a code is able to follow itself. This is due to the fact that the videos were coded using an incident-by-incident method, as recommended for use with observational data (Charmaz, 2006) such as these videos. In some cases, two *Suggestions*, for example, were coded subsequently. This is because the researcher coding these viewed the two incidents of *Suggestion* as ones with a different context (i.e. an initial conversation would have the pair suggesting ways of fixing an error, but they would then move on to suggesting a method refactor; or the driver would start to make a suggestion, but be interrupted by the navigator, who is suggesting something different). In most cases, these subsequent codes do not occur frequently, but Figure 23 shows that there is an 18.5% chance of a *Suggestion* following another *Suggestion*.

Those codes which occur more than expected by chance (i.e. higher than 14.29%) are summarised in Table 17.

Table 17: A list of most common transitions for each analytic code

Code	Common Transition	Chance of Occurrence
Explanation	Suggesting	38.1%
Code Discussion	Suggesting	41.0%
Muttering	Suggesting	37.3%
	Code Discussion	27.1%
Unfocusing	Review	23.5%
	Silence	22.1%
	Suggesting	19.1%
Review	Suggesting	33.3%
	Silence	18.8%
	Explanation	17.4%
Silence	Suggesting	50.0%
	Review	16.2%
Suggesting	Silence	21.9%
	Muttering	21.9%
	Suggesting	18.5%

3.6.2 **A Visual Representation of Code Transitions**

Figure 24 presents a visual representation of Table 17, illustrating the most common transitions to follow each state.

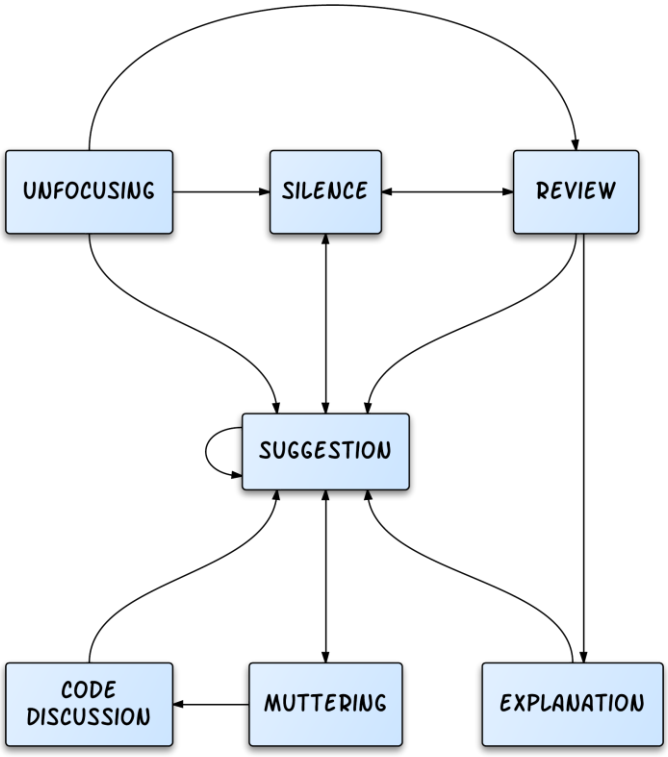


Figure 24: A visual representation of the most common state-to-state transitions

Figure 24 provides an easily understandable description of the most typical communication flow exhibited within the *pairwith.us* partners. Each node in the diagram represents a different communication state for the pair to be in. For example, if the pair is in an *Unfocusing* state, this is most commonly followed by a *Review* stage, *Silence*, or a *Suggestion*.

3.6.2.1 Codes that lead to an Unfocusing State

Thus far, the discussion has been centred around transitions from analytic codes; that is, which codes tend to *follow* a specific code. This works well with most codes, and is evident in Figure 24 (e.g. *Explanation* follows *Review*). However, it can be seen that

Unfocusing is the only code which does not seem to follow any other codes (in more technical terms, the *Unfocusing* node does not seem to have an entry point) in the data presented above.

In Figure 18 above, it can be seen that *Suggestion*, for example, is the most common code following a *Code Discussion* state (41%). *Unfocusing* has also been seen to follow this state – but only has a 15% chance of occurring. This comparatively low probability rate, combined with the fact that *Unfocusing* transitions make up only 8% of all transitions, leads to *Unfocusing* not appearing to be follow the *Code Discussion* state (or any other code) in the list of 'most frequent' transitions.

Despite this transition not being one of the most common, it would still be worthwhile to explore transitions that lead to this code to obtain an insight into typical reasons for the pair entering this state, and breaking their focus. The *pairwith.us* data was analysed and displayed in a similar way to Figures 17 to 23 above, to chart the occurrence of each code leading to an *Unfocusing* state. The resulting data is shown in Figure 25.

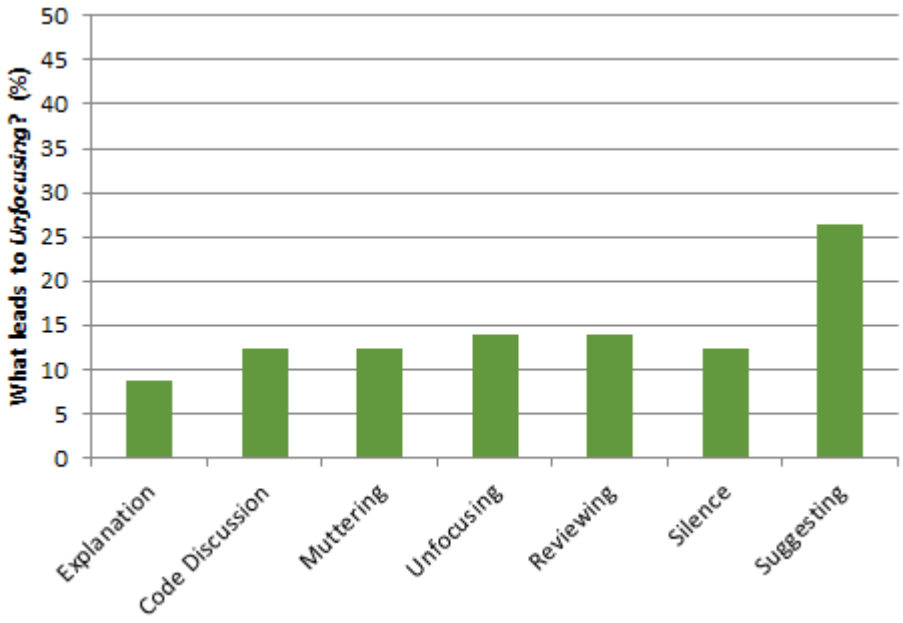


Figure 25: What codes lead to Unfocusing?

It can be seen that 26% of all *Unfocusing* states were preceded by a *Suggestion*. The remaining codes have a lower frequency of preceding *Unfocusing* with all but one being in the range of 12% to 13%. *Explanation*, at 8%, is the code with the lowest probability of doing so. A typical transition from a Suggestion to Unfocusing is shown in Figure 26.

N: "So we do a new class-"
D: "-Like so."
N: "Instead of putting it in jNarrate, put it in FitnesseNarratives. It's the WebUserTherapist."
D: "Oh. 'Therapist', not 'Terrapin'."
N: "So it's not just me that sees 'Terrapin' when you start writing 'Therapist'?"
D: (laughs)
N: "I was honestly looking it and had a picture of a terrapin in my head."

Figure 26: A transition from Suggestion to Unfocusing

The *pairwith.us* team were asked to comment on their personal experiences with ‘unfocusing’ states shortly after the coding scheme was first proposed. They said that “keeping the mood high with jokes, breaks, etc. is quite important to being able to maintain such intense focus for long periods of time”. This matches what is evident from the conversation fragment above: the pair is initially working on code and suggestions posited by the navigator. Once the driver makes a typing error, this conversation quickly turns into more of a joke, which prompts an unrelated discussion on terrapins. Once the discussion finishes, the pair resume their focused programming activity.

3.7 Limitations

This chapter discussed the analysis of a set of pair programming videos, and the subsequent creation of a coding scheme. There are several limitations, which this section will consider.

The analysis leading to the coding scheme, as well as the subsequent discussion on transitions between states, was based on observations of the same pair of developers. Whilst this means that the findings thus far are not generalizable, it does allow for an initial understanding of how this pair experiences various communication states. At this stage, the coding scheme needs to be tested with other pairs before it can be generalised.

A further limitation is that the transcripts and coding presented in this chapter were carried out by a single researcher. It is possible that this may lead to bias, and impact on the findings presented so far. Confidence that this was not the case is gained, however, by the fact that the inter-rater reliability values for the coding scheme are high.

Finally, the observed pair were responsible for recording their own videos, and therefore could have either "performed" for the camera, or otherwise edited the videos. All videos were carefully scrutinised for any such instances. Some of the early videos contained sections where the pair used social media to engage with their viewers, or read out e-mails and comments from visitors to their website, but this practice was discontinued in later videos. The later videos were chosen as candidates for further analysis.

3.8 Summary

This chapter describes the detailed analysis of a set of videos produced by an expert pair, and the creation of a coding scheme, using an approach informed by grounded theory through open coding, the construction of codes and comparison of the data. The resulting coding scheme was refined through multiple iterations of the analysis process and confirmed by an inter-rater reliability test. This coding scheme was then defined, consisting of the following states: *Explanation*, *Code Discussion*, *Muttering*, *Unfocusing*, *Review*, *Silence* and *Suggesting*. The way it was used to code the *pairwith.us* videos was then analysed further, to generate preliminary usage data which led to an understanding of common transitions that occur between communication states.

Chapter 4: Confirmative Studies

This chapter describes confirmative studies that were carried out with eleven expert pairs from two different industry sectors. These studies were carried out to investigate the relevance and generalisation of the verbal communication codes discussed in the previous chapter. The chapter concludes with a review and re-examination of the analytic code transitions, and the generation of communication patterns and guidelines which will become the main focus of this thesis.

Confirmatory research typically tests *a priori* hypotheses, which are made prior to any measurement, and derived from results of previous studies (Jaeger and Halliday, 1998). It is the next step after the gathering and analyses of data, and culminates with inductive inferences.

This work focuses on the analytic codes and transitions following research carried out with the *pairwith.us* material. Following the grounded theory-inspired approach outlined in Chapter 3, iterations over the data and the associated codes need to reach a point of convergence in order to complete. The coding scheme will be therefore tested across a broader sample of expert pairs in a more authentic setting, to determine the degree to which the set of codes generated so far can be generalised.

4.1 Method

To build on the work of the previous chapter, further video footage of expert pairs was required to extend the observation and analysis of programming sessions from the previous *pairwith.us* context to a broader spectrum of developers. This study first required making contact with multiple expert pairs who had been working within one or more pairs in industry (i.e. practising various agile software methodologies and pair

programming, in particular, as their main occupation) for a minimum of six months. Video capture and other observations should take place in the workplace: conducting the observations in a natural setting would ensure that the behaviours observed are as close to typical for the pair as possible (Preece et al., 2011).

The purpose of the observations was to gather further data on verbal communication within expert pair programming and to verify whether the analytic codes and transitions discussed in the previous chapter could be applied to a wider set of expert pairs.

4.1.1 **Participants**

In order to recruit companies to participate in the observation sessions, personal contacts were first tried in order to establish relationships with local companies. It was found that many companies in the area did not practise agile, or if they did, they practised a watered-down version of agile that met their needs (e.g. only pair programming once a week, or using the term 'pair programming' to mean 'asking for advice'). Others did not wish to participate.

A wider net needing to be cast, a leaflet was produced with the aim of recruiting expert pairs (see Appendix E) and made available on social media (e.g. Twitter, Facebook and LinkedIn) and several mailing lists (e.g. BCS-SPA), with recipients encouraged to share, print, and distribute. This document briefly described the research aims of the study, and also encouraged pairs to sign up for observation and recording sessions.

Several companies that distributed the leaflet internally showed initial interest in the study – however, due to the presence of video and audio recording devices, chose not to proceed with the observation. A large number of companies were contacted; two London-based companies agreed to participate, agreeing to the conditions stipulated.

The observations involved 11 pairs across two different industrial sectors:

- Company 1 (C1) is a company which focuses on delivering high quality broadband and telephony around the UK. The team at C1 use agile methodology constantly, implementing practices such as scrum and Extreme Programming.

  Following a morning scrum, each programmer at C1 is allocated a task, and chooses a pairing partner based on the expertise required to finish that task.

- Company 2 (C2) is one of the leading global technology platforms for social video distribution and analytics. Several teams within C2 use agile practices to continuously test and develop their technology.

  Following a daily scrum, each task is allocated to a specific pair by the scrum master, depending on the individual programmers' skill set.

4.1.2 **Procedure**

Typically, a complete observer refrains completely from interacting with people (Gold, 1957, Bryman, 2012), choosing instead to use methods of observation that are as unobtrusive as possible. Therefore to minimise intrusion, the researcher acted as a complete observer from a distance. This also minimised the disruption to the pair's working output.

The observation procedure for each pair was as follows:

- The researcher was initially introduced to the team, and asked that there be no predefined schedule for observation to allow for as natural a setting as possible, so that each pair would not be anticipating any set disruption. It was agreed that following each observed session, the team leader would select one of the

available pairs for the next observation. The researcher was also given access to film pairs at their normal workstations, rather than in a separate area.

- The researcher introduced himself to the pair and briefly discussed the main aims of the observation. Following this, the pair was asked to sign individual consent forms (Appendix E).

- The recording equipment was set up by the researcher, and filming duration was agreed with the pair (typically an hour). In order to keep the natural flow of interactions and in order to have as discreet a setup as possible, the camera was placed behind the participants. The researcher reinforced the possibility of the recording equipment being switched off at any point if desired during the session. At this point, the researcher would leave the pair until the session was over.

- Following the recording session, the researcher explained the research aims in more detail, and answered any questions that may have arisen during the observation session. The pair was then asked to individually and anonymously fill in forms (Appendix E) relating to their experience and confidence with pair programming. The results of this are reported in section 4.2.1.

For the duration of the actual session, no contact was planned. Each session was recorded using minimal equipment designed to be as unobtrusive as possible, as the authenticity of the session was deemed to be of importance:

- A video camera would be set up behind the pair (outside their field of vision);

- A portable audio recorder would be set up behind the pair's monitor/s.

The use of screen-capture software and webcams to capture images was rejected since participants were industry employees dealing with commercially sensitive data and working on company machines.

4.1.3 Issues with Observations

Following both observation sessions, the researcher transferred the raw video data (n=11) to a PC for further analysis. Due to the fact that all recordings were done in open offices, at the participant's usual workstations, several videos suffered from high levels of background noise. Furthermore, the participants were sometimes very quietly spoken, and their speech was therefore not fully picked up by the recording equipment.

The issues presented above could have been avoided by conducting sessions in a quieter environment. However, this would have detracted from the naturalistic setting that was observed. By being at their workstations, pairs were able to discuss their problem with other developers or leave their desk for an extended period of time due to other programmers' issues, and behave as they would on a typical day.

Another perceived solution would have been to provide the pairs with wearable microphones. However, it was felt by the researcher that this would have been too intrusive for the pair, and might have impacted upon their behaviour, and hence, their captured communication. Furthermore, it was not known whether providing microphones would have impacted the office dynamic: with the existing set-up, if an external developer wanted to discuss a private matter, they would ask the pair to move away from the camera, as the camera was visible. Had the pair been wearing microphones, an external developer to the pair might not have known to ask for the microphones to be removed, or be switched off, which would be an ethical issue.

A further issue was uncovered when transcribing the videos: due to the camera being set up behind the participants, it was difficult to distinguish the current speaker in certain videos. Two types of transcript were therefore created depending on the speakers' clarity in each video: one where the speakers, and therefore their individual communications, are clearly identified; and one where the speakers could not be identified, and are therefore not listed. Each transcript was fully coded using the coding scheme discussed in Chapter 3. The incident-by-incident style of coding used emphasised on what was being said, rather than who it was that said it.

4.2 Data Analysis

All data was gathered and analysed in a process informed by grounded theory as a continuation of the analysis that was discussed in Chapter 3. The next section presents results and discussion a detailed discussion analysing the participants' previous experience with both solo programming and pair programming.

This will be followed by a more detailed data analysis stage consisting of two stages: coding and transitions. The data is compared with the results from Chapter 3 to ascertain agreement and to understand any changes that need to be incorporated into the coding scheme as a result of using them in an industrial setting.

4.2.1 **Participant Experience**

Following the observation sessions, each participant was asked to fill in surveys related to their previous programming experience, and to the typicality of the observed pairing session (Appendix E). The results of these surveys are presented here.

Company 1

A total of six pairs (all male) were observed at C1, with each session lasting roughly one hour. Individually, the developers (n=12) reported industrial pair programming experience of 4.92 ± 2.30 years. When asked to specify how long each developer had collaborated with the observed session's pairing partner, participants reported an average experience of 1.22 ± 0.75 years.

Post-study, the following statistics were gathered on a 5-point Likert scale, with 1 indicating a low agreement and 5 indicating a high agreement for the following statements:

- *I feel pair programming is more beneficial than solo programming* was rated 4.4 ± 0.67.
- *During this session, I found communicating with my partner to be easy* was rated 4.6 ± 0.51.

The numbers reported above indicate that the observed pairs felt they had displayed a good standard of communication, and that they believed that pair programming was largely more beneficial than traditional programming methods.

The researcher asked each pair to rate how typical the observed session was (when compared with other pair programming sessions that the pair had participated in) on a similar 5-point Likert scale. The rating for this was 3.92 ± 0.79, with the developers pointing out that each pair programming session was likely to be different due to reasons related directly to the problem at hand, such as: *"This [problem] was a very technical and abstract one, which is why our session was not typical"*.

Company 2

A total of five pairs (four of which were both male, and one of which was mixed) were observed at C2, with each session lasting roughly one hour. Individually, the developers (n=10) reported industrial pair programming experience of 2.02 ± 1.79 years. When asked to specify how long each developer had collaborated with the observed session's pairing partner, participants reported an average experience of 0.61 ± 0.75 years (*Mdn* = 0.5 years).

In this last statistic, the standard deviation was greater than the mean. This occurs as the data comes from a small sample size (10 developers) with outliers, thus leading to an abnormal distribution of the data. The median value is provided in the brackets following the data, as this is more robust against abnormally distributed data sets.

Post-study, the following statistics were gathered on a 5-point Likert scale, with 1 indicating a low agreement and 5 indicating a high agreement for the following statements:

- *I feel pair programming is more beneficial than solo programming* was rated 4.2 ± 0.63.
- *During this session, I found communicating with my partner to be easy* was rated 4.3 ± 0.48.

This indicates that similarly to the pairs observed at Company 1, the pairs reported a good standard of experienced communication and believed that pair programming was more beneficial than traditional programming methods.

The researcher asked each pair to rate how typical the observed session was (when compared with other pair programming sessions that the pair had participated in) on a similar 5-point Likert scale. The rating for this was 4 ± 0.47, indicating that all observed sessions were largely typical of standard pair programming sessions.

One of the pairs indicated that they had spent a large amount of time discussing previously written code, and planning possible courses of action. The pair indicated that they still considered these actions to be pair *programming*, as they were "setting the groundwork" for tasks that were implemented after the observation.

4.2.2 Coding the Videos

The captured videos were transcribed and coded. As a result of the recording issues discussed in section 4.1.4, not all videos could be successfully transcribed. The videos that were deemed of poor quality (n=5) were subsequently discarded from the study.

Using Transana to transcribe and code the remaining videos (three from Company 1, and three from Company 2) allows the resulting data to be gathered in a similar way to the data gathered in Chapter 3, allowing for a fairer comparison of the data. The approach used is similar to the one carried out in the previous chapter: each video was imported into Transana, and each sentence of the transcribed conversation was time-stamped and linked with the video file for retrieval. Each video was then coded using the coding scheme described in section 3.4; comparisons were made between instances of incidents rather than individual lines of the transcript (Charmaz, 2006).

4.2.3 Inter-Rater Reliability

Two colleagues from the School of Computing at the University of Dundee were asked to perform an assessment of the coding scheme's inter-rater reliability (IRR), when

applied to the videos obtained from Company 1 and Company 2. Colleagues were recruited from different research groups within the School, with no ties to the study.

Initially, the raters were provided with the coding scheme in section 3.4. The researcher selected a sample of videos recorded from both Company 1 and Company 2, and asked each rater to code a subset of video from the sample chosen. An IRR was performed to determine consistency among the raters, including the researcher.

The individual reliability was first calculated using Cohen's Kappa (Cohen, 1960). The reliability between the researcher and rater A was Kappa = 0.798 ($p < 0.001$), 95% CI (0.603, 0.873). The reliability between the researcher and rater B was Kappa = 0.796 ($p < 0.001$), 95% CI (0.663, 0.937). The reliability between rater A and rater B was Kappa = 0.715 ($p < 0.001$), 95% CI (0.456, 0.758).

The IRR for all raters, calculated using Fleiss' Kappa (Fleiss, 1971), was found to be Kappa = 0.768 ($p < 0.001$), indicating a substantial agreement (Landis and Koch, 1977).

4.2.4 **Results: Coding**

In order to understand how the pairs exhibited each instance of analytic codes and make comparisons with the *pairwith.us* data, the total number of occurrences for each analytic code over the six videos was first calculated.

A percentage value, depicting the number of times each code occurred within each video, is given in Table 18.

Table 18: The number of occurrences (percentage value) for each analytic code

Analytic Code	Code (%) in C1 Video #1	Code (%) in C1 Video #3	Code (%) in C1 Video #5	Code (%) in C2 Video #3	Code (%) in C2 Video #4	Code (%) in C2 Video #5
Suggesting	28.9%	34.2%	38.1%	18.1%	26.7%	28.8%
Explanation	15.6%	11.2%	14.3%	25.9%	17.8%	14.4%
Unfocusing	5.6%	2.1%	1.2%	11.2%	3.3%	3.2%
Silence	10.0%	8.0%	13.1%	6.9%	8.9%	8.8%
Muttering	10.0%	17.1%	13.1%	6.9%	11.1%	16.8%
Reviewing	18.9%	12.8%	11.9%	27.6%	22.2%	12.0%
Code Discussion	11.1%	14.4%	8.3%	3.5%	10.0%	16.0%

It can be seen that *Suggesting* is the activity that occurs the most across all videos, with *Unfocusing* occurring the least amount of times.

The data presented above is similar across all videos, with some differences in the data in Video 3 from Company 2. The pair observed here were engaged in a review of previous code, rather than actively working on solving a problem. Due to this, there are higher amounts of *Reviewing* and *Explanation*, and lower amounts of *Suggesting* and *Code Discussion.*

The entire set of occurrence values within the videos was added up per company, with a set of mean percentage values calculated. The average proportion of occurrence of each activity is presented in Table 19 for comparison with the *pairwith.us* data discussed in Chapter 3.

Table 19: Occurrence percentage values across all three contexts

Analytic Code	Occurrence percentage values		
	pairwith.us	**Company 1**	**Company 2**
Suggesting	32%	35%	25%
Explanation	9%	14%	19%
Unfocusing	10%	2%	6%
Silence	11%	11%	8%
Muttering	12%	14%	12%
Reviewing	15%	13%	20%
Code Discussion	11%	11%	10%

4.2.5 **Discussion: Coding**

The occurrence values were compared with values from the *pairwith.us* context discussed in Chapter 3. *Suggesting* is the code that occurs the most often in all three cases. The codes for *Muttering*, *Silence* and *Code Discussion* are similar across all three settings.

There are some differences within the other codes in the industrial setting (C1 and C2), highlighting a greater number of *Explanations*, and fewer instances of *Unfocusing*.

One interpretation of these differences is the contrasting settings. Whereas the observations from industry were all entirely workplace-based, the *pairwith.us* team recorded their videos as their own personal hobby. The decrease in 'off-topic' discussions could be due to the fact that the *pairwith.us* team have no set deadlines and thus have the leisure of taking several breaks whilst pairing. In contrast, in the work environment, the pairs in Company 1 and Company 2 were more focused on their

deadlines, and thus spent less time conversing and more time focused on the task at hand.

An alternative interpretation is that in both industrial circumstances, pairs were decided daily, with allocations depending on the developers who had the skills to solve particular problems. This meant that some developers without a strong familiarity with the code they were working on had a greater need for explanations, whereas at *pairwith.us*, the same pair was constantly working on the same piece of code – and therefore had a higher affinity with it, and less need to constantly explain certain functions.

It can be seen that the coding scheme developed from the pairwith.us data in Chapter 3 was successfully used to code and analyse data from other industry-based pairs. The analytic codes created from the observation and analysis of verbal interactions within the pairwith.us pair have been successfully applied to a number of pairing sessions where pairs from the industry have been observed at two different workplace circumstances.

Grounded theory methodologies typically state that studies should collect data until the point of saturation is reached. Selden (2005) explains the process of saturation as follows: "One keeps on collecting data until one receives only already known statements." This is the case here, as confirmed by the IRR – the analysis was satisfied with the existing codes.

Following the successful analysis of six industry-based pairs, the coding scheme is seen to be robust, and is a solid basis on which to continue this research.

4.2.6 **Results: Transitions**

As in Chapter 3 with the *pairwith.us* data (section 3.6.1), the communication flows from one activity to the next were analysed for both Company 1 and Company 2.

Figure 27 – Figure 33 show the occurrences of code transitions for each of the seven codes (as percentages) for each company. As there are seven codes, the mean value is 14.29%; this is depicted on each chart as a horizontal line. The most common transitions in the data are identified as all occurrences that were higher than this number. These are summarised in Table 20.

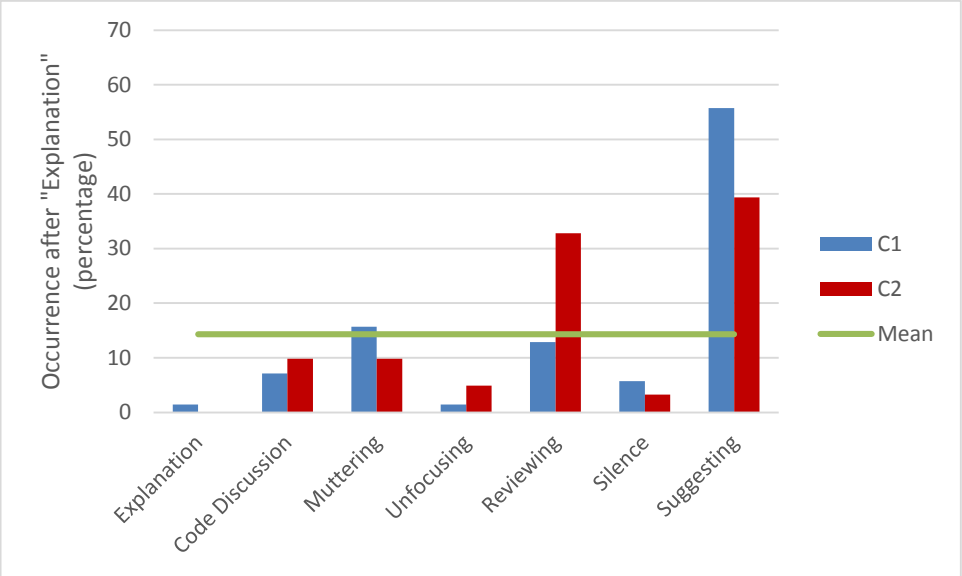


Figure 27: Codes that followed "Explanation" in the observations from C1 and C2

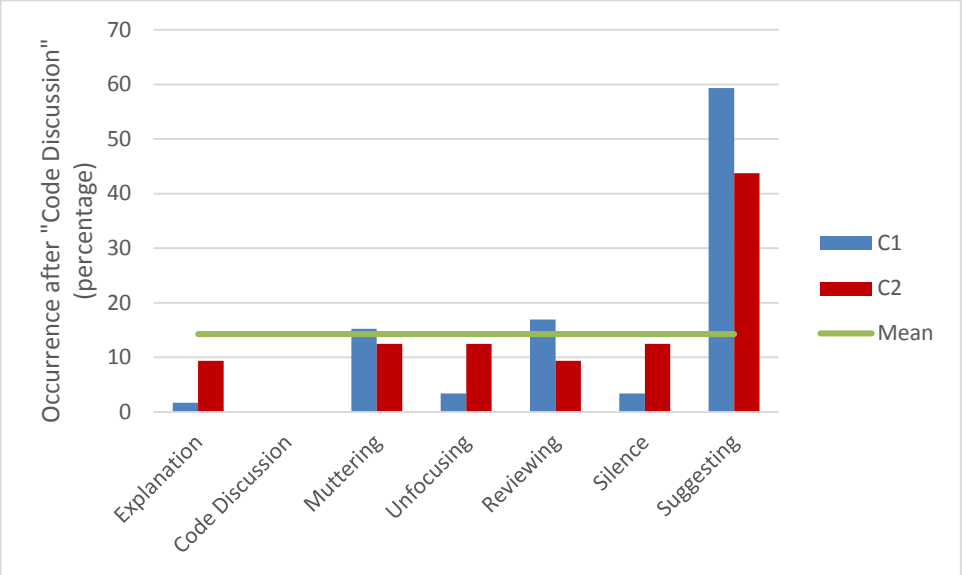


Figure 28: Codes that followed "Code Discussion" in the observations from C1 and C2

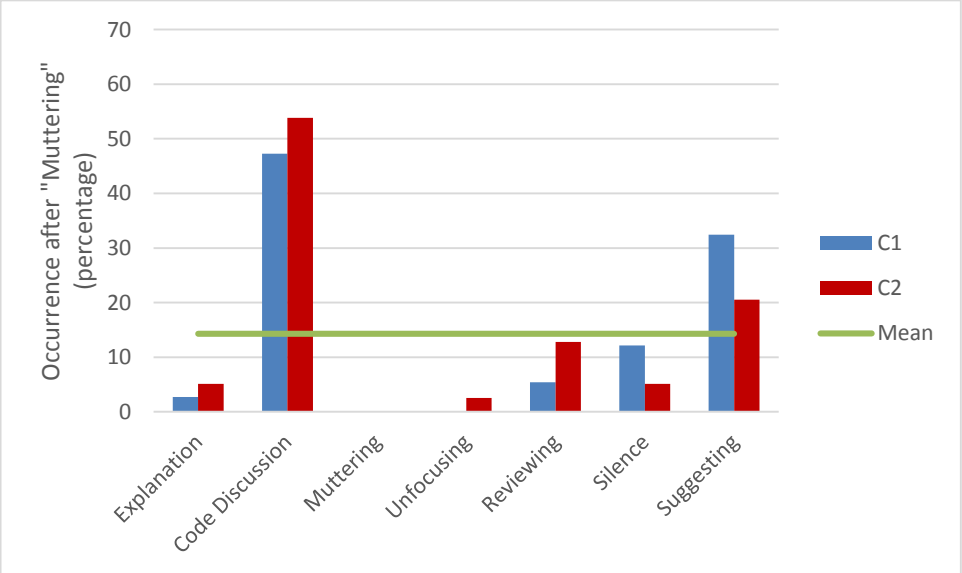


Figure 29: Codes that followed "Muttering" in the observations from C1 and C2

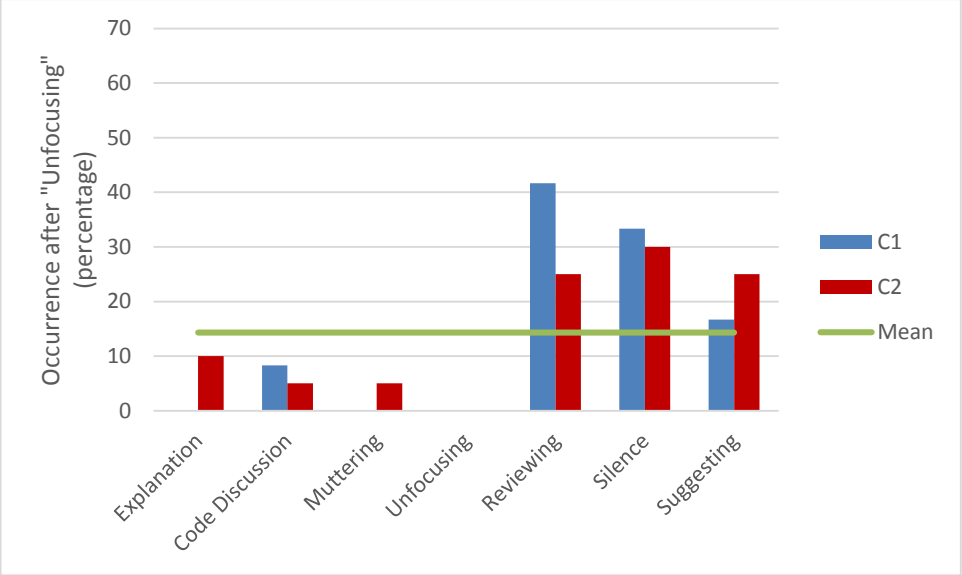


Figure 30: Codes that followed "Unfocusing" in the observations from C1 and C2

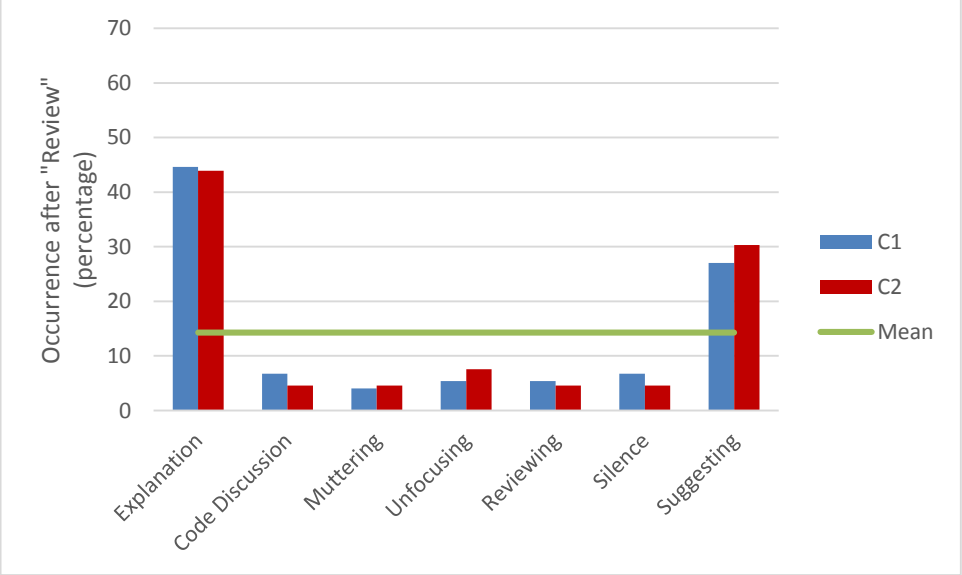


Figure 31: Codes that followed "Review" in the observations from C1 and C2

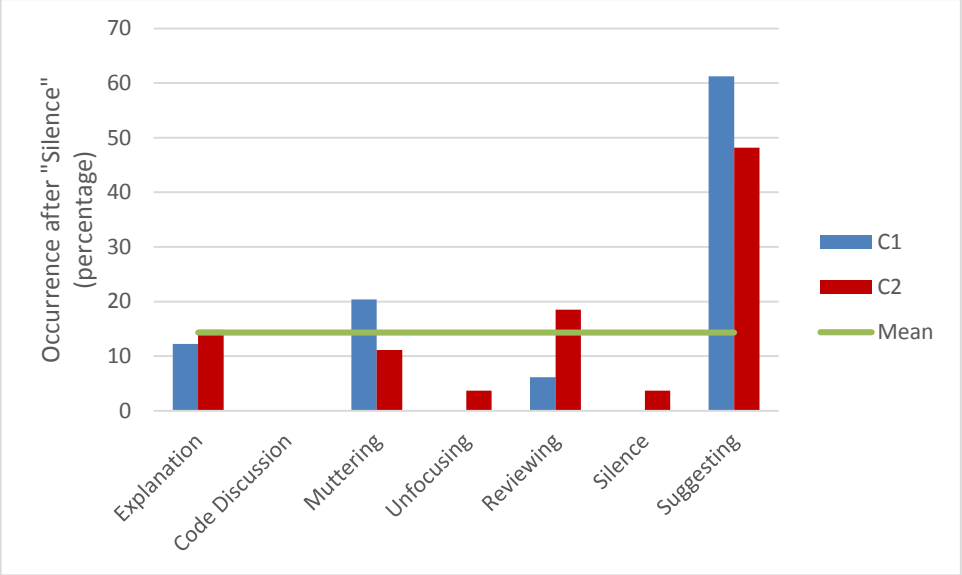


Figure 32: Codes that followed "Silence" in the observations from C1 and C2

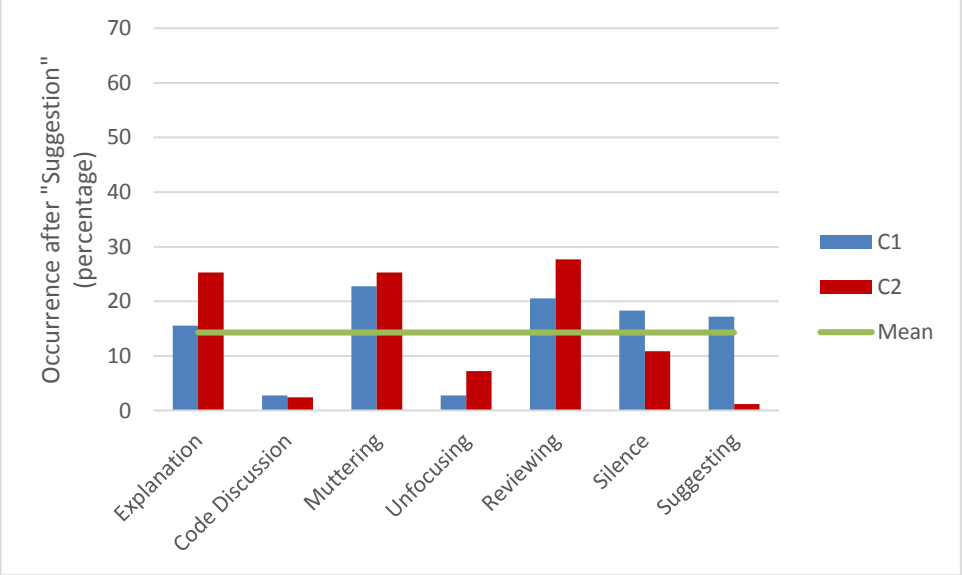


Figure 33: Codes that followed "Suggesting" in the observations from C1 and C2

Those codes which occur more than expected by chance (i.e. higher than 14.29%) are summarised alphabetically in Table 20.

Table 20: The most common transitions for each analytic code

	Code is most commonly followed by…		
	C1	**C2**	***pairwith.us***
Explanation	Muttering Suggesting	Reviewing Suggesting	Suggesting
Code Discussion	Muttering Reviewing Suggesting	Suggesting	Suggesting
Muttering	Code Discussion Suggesting	Code Discussion Suggesting	Code Discussion Suggesting
Unfocusing	Reviewing Silence Suggesting	Reviewing Silence Suggesting	Reviewing Silence Suggesting
Review	Explanation Suggesting	Explanation Suggesting	Explanation Silence Suggesting
Silence	Muttering Suggesting	Explanation Reviewing Suggesting	Reviewing Suggesting
Suggesting	Explanation Muttering Reviewing Silence Suggesting	Explanation Muttering Reviewing	Muttering Silence Suggesting

The Company 1 and Company 2 transitions were reviewed with each other, and with the *pairwith.us* data from Table 17 in Chapter 3. A discussion about the similarities and differences between the settings is given next.

4.2.7 Discussion: Most Common Transitions

Most of the common transitions (e.g. *Explanation* to *Suggesting*) were observed to have occurred in all contexts. Some minor differences are evident when comparing data between industry-based pairs (C1 and C2) and the *pairwith.us* team. Several transitions are missing in the industrial context – for example, *Review* does not commonly lead to *Silence*. Most notably, *Suggesting* is also followed by *Explanation* and *Reviewing*.

The differences here are mostly based on the reduction of off-topic instances in the observed environment, as discussed above. The addition of *Explanation* as a follow-up to *Suggesting* is quite notable, implying that pairs within the industry were more likely to suggest a next step and to also explain how and why that suggestion was a positive step forward. This is likely due to the fact that whereas the *pairwith.us* team worked together for a length of time, the pairs at C1 and C2 changed daily, and therefore, there was no guarantee of previous familiarity with the code.

4.3 Updating Transitions

Descriptive statistics were generated by calculating the means and standard deviations for each code transition possibility between the three contexts (Company 1, Company 2, and *pairwith.us*). These are presented in Table 21 below, which is meant to be read in landscape orientation as follows: *The percentage of occurrences of* ***row*** *leading to* ***column*** *is* ____.

The mean value for each row in Table 21 was calculated. Any value that was higher than the mean (14.29%) was considered to occur “more than average”. These values are shaded in the table.

Table 21: Probability of transitions between codes (all three settings)

	Explanation (%)	*Code Discussion (%)*	*Muttering (%)*	*Unfocusing (%)*	*Reviewing (%)*	*Silence (%)*	*Suggestion (%)*
Explanation	4.7 ± 6.96	7.8 ± 1.83	10.6 ± 4.73	4.8 ± 3.26	20.0 ± 11.12	7.8 ± 5.78	44.4 ± 9.83
Code Discussion	7.5 ± 5.15	2.7 ± 4.73	12.0 ± 3.56	9.7 ± 5.44	12.6 ± 3.91	7.5 ± 4.62	48.0 ± 9.89
Muttering	5.4 ± 2.90	42.8 ± 13.93	0.0 ± 0.00	5.4 ± 7.20	10.6 ± 4.51	5.8 ± 6.10	30.1 ± 8.63
Unfocusing	4.3 ± 5.14	8.4 ± 3.38	4.6 ± 4.42	3.9 ± 6.79	30.1 ± 10.08	28.5 ± 5.79	20.3 ± 4.28
Review	35.3 ± 15.52	5.7 ± 1.11	4.8 ± 0.90	8.2 ± 3.14	5.7 ± 1.38	10.1 ± 7.70	30.2 ± 3.15
Silence	13.5 ± 1.29	3.6 ± 6.24	10.5 ± 10.22	4.4 ± 4.77	13.6 ± 6.60	1.2 ± 2.14	53.1 ± 7.07
Suggesting	17.4 ± 7.20	5.3 ± 4.67	23.3 ± 1.76	6.1 ± 2.98	18.5 ± 10.36	17.0 ± 5.65	12.3 ± 9.65

Some of the lower values are seen to have a standard deviation that is higher than the mean (e.g. the overall probability of an *Explanation* leading to a second *Explanation* is

4.7 ± 6.96). This indicates that there was a high degree of variability between the numbers used to calculate the mean, and therefore, any such values were excluded from consideration for further analysis.

Table 21 and the bar charts above (Figure 27 - Figure 33) were used to update the *pairwith.us* transitions diagram (Figure 24), resulting in Figure 34.

4.3.1 Revising the Transitions Diagram

Figure 24 in Chapter 3 depicted the most common transitions as identified by the *pairwith.us* data. This depiction can be updated with the additional data from the industry-based pairs. The revised diagram is given in Figure 34. As before, each code represents a communication state which the pair is experiencing at any given time:

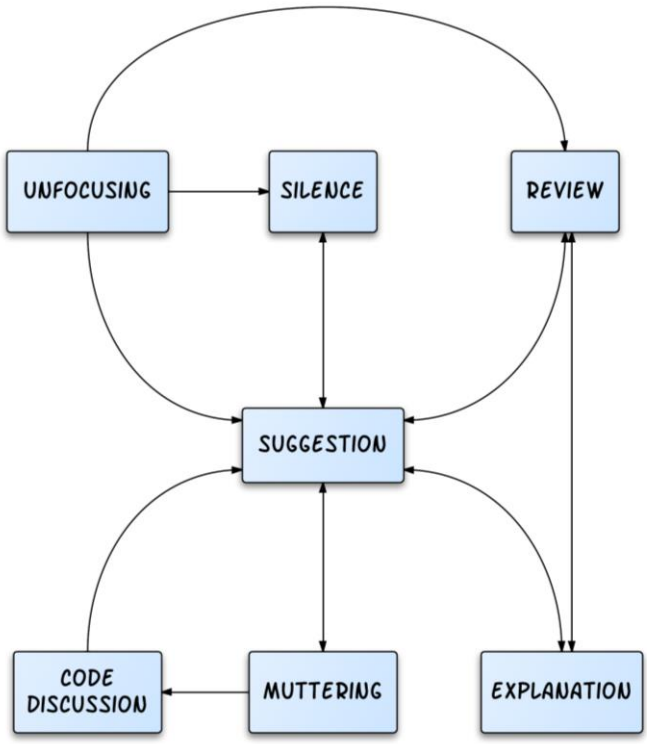


Figure 34: Most common transitions between codes (all three settings)

4.3.1.1 Code that Lead to an Unfocusing State

As before, all states have at least one 'entry' and 'exit' except *Unfocusing*. Across all three contexts, *Unfocusing* makes up approximately 6% of all transitions. When calculating the 'most common' transitions, data that shows what precedes *Unfocusing* is comparatively low when compared to data from other transitions. The data is hence not displayed in the transitions diagram seen above, making it seem as if no codes lead to *Unfocusing* – when in fact, it is simply the probability of reaching an *Unfocusing* state in the first place that is low.

An understanding of what transitions did lead to *Unfocusing* could provide an insight into typical reasons for the pair choosing to break their focus. Figure 35 shows the proportions of codes that led to an *Unfocusing* state in the context of C1 and C2.

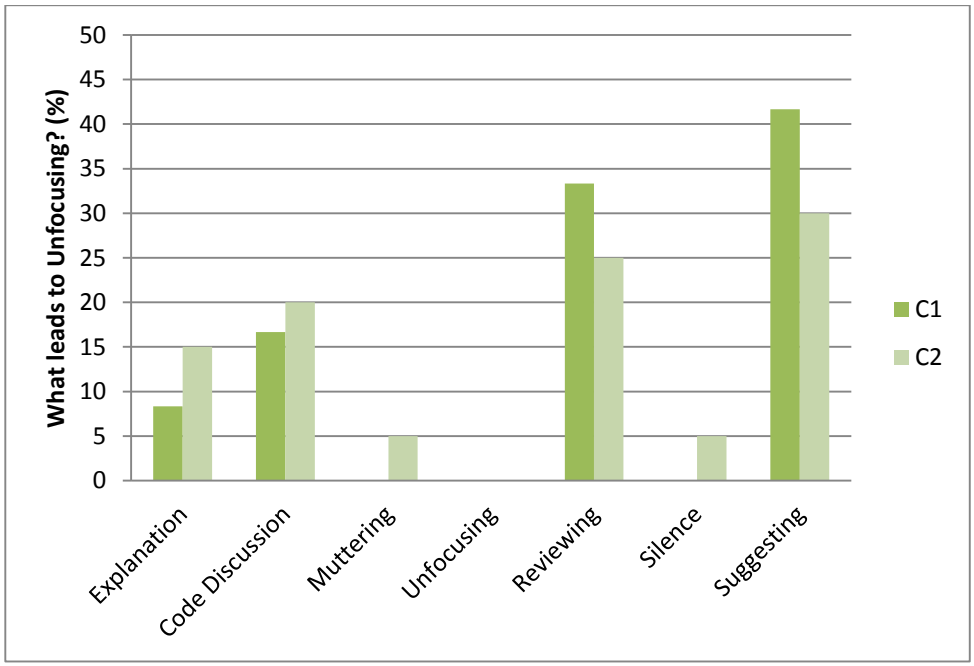


Figure 35: What codes lead to Unfocusing?

Typically, an *Unfocusing* state can be seen to be initiated in one of two ways:

The first (Figure 36) is when the pair are discussing their current tasks, but are hitting road blocks (e.g. they cannot work out the correct solution whilst suggesting possible courses of action, or cannot understand which part of the examined legacy code is to be actioned). This leads the pair to actively choose to break their focus, as per the following conversational snippet:

D: I think it's worth keeping it.

N: I can see why you're… you know, looking. The generics just get crazy.

D: Do you wanna bail?

N: Yeah, let's bail – I'll think about it in my own time.

Figure 36: Initiating an Unfocusing state

The second way of initiating an *Unfocusing* state (Figure 37) is less active, and can be seen when a pair is actively working and having task-related discussions. On occasion, the conversation veers off-topic, and thus the pair lose focus. In the following example, a member of the pair expresses frustration at the code and mentions a holiday he had just returned from. This prompts his colleague to ask about the holiday, thus initiating a discussion that was not relevant to the current task.

N: Can we repeat the same thing for #170? Just to have a look...

D: This is taking latency to a whole new level.

N: I might just go back on holiday!

D: Oh yeah – how was it?

Figure 37: Initiating an Unfocusing state

It is interesting to understand and explore the different ways in which an *Unfocusing* state can be initiated. This could potentially help novice pairs be more aware of when they are about to experience an *Unfocusing* state, thus allowing them to choose whether to actively break, or keep, their focus.

Exploring and analysing how the codes *follow* each other can give insight into the conversation flow that the pair is experiencing, and also gives information about which communication state is most likely to follow. These transitions were analysed with the aim of guiding first-time pairs by having them understand or recognize their current state, and identify possible next actions. Looking forwards, in this case, is likely to have more utility than looking backwards. However, understanding which codes are most likely to *precede* each other can give information with which to better understand this conversation flow, as seen in this section discussing *Unfocusing*, and could lead to understanding whether any states can be (or should be) prevented. This analysis will be discussed further in the 'future work' section of Chapter 8.

4.4 Proposed Guidelines

This thesis centres on a research question: *can extracted communication patterns from expert pair programmers be used to help novice student pairs to improve their intra-pair communication?*

In Chapters 3 and 4, the first part of this research question was tackled: communication patterns were extracted from expert pair programmers (Figure 34). In their current form, the patterns have no context or definition, and are therefore not sufficient to convey useful information to their target audience: these patterns need to be re-cast into guidelines.

A *guideline* is, by definition, a general rule or a piece of advice, synonymous with a recommendation or a suggestion: "an indication of a future course of action"[8]. Existing guidelines for pair programming have been discussed in Chapter 2, showing that currently, apart from a paper by Williams et al. (2000), there are few guidelines that aim to advice novice pairs on how to pair program, without any focus on how to communicate whilst pair programming.

The following section discusses certain conversational patterns within the larger context of the transitions diagram (Figure 34) and extracts guidelines from these patterns.

4.4.1 Extracting Patterns and Generating Guidelines

In order to better understand the transitions depicted above, Figure 34 was segmented into several subsections, to achieve the following:

i) To understand what happens following an *Unfocusing* event, and how this leads to the pair regaining focus;

[8] http://dictionary.reference.com/browse/guideline/

ii) To extract repeated communication behaviour.

Each subsection depicted different stages of the communication process within pair programming. Each subsection is referred to as a 'pattern', representing the different communication states a pair can experience, and the various ways of transitioning between these states. Each extracted pattern was used to form the basis for a set of guidelines.

Whereas each code represents a different communication state for a pair, each pattern represents different stages of the pairing process. Patterns can illustrate a whole set of communication states describing, for example, a reviewing cycle, or actions leading to the pair deciding to take a break from their current task.

Figure 34 was segmented into three patterns: one that looks at all possible outcomes from an *Unfocusing* state; and two which consider certain repeated behaviours. The identified patterns are:

1. A pattern linking *Unfocusing*, *Review*, *Silence* and *Suggestion* on the top half of the diagram, explaining actions that follow an *Unfocusing* event. This is called the Restarting Pattern;

2. A pattern linking *Review*, *Explanation* and *Suggestion* on the right-hand side of the diagram, showing repeated communication behaviour. This is called the Planning Pattern;

3. A final pattern linking *Muttering*, *Code Discussion* and *Suggestion* on the bottom-left of the diagram, showing repeated communication behaviour. This is called the Action Pattern.

Instances of each pattern were observed in the pair videos and reviewed in order to explore why and how certain patterns were being exhibited. At this stage, the *pairwith.us* team was consulted about the existence of these patterns. They confirmed that these were behaviours that they recognized. Discussions with the *pairwith.us* team and a member of teaching staff (with experience of teaching agile) within the School of Computing were used to identify guidelines and to structure these in a suitable way for educational purposes.

The pair programming communication guidelines were therefore created to give users more insight into the instructions offered by these patterns. The aim of the research is to investigate whether providing novice pairs with communication patterns from expert pairs will allow them to improve their intra-pair communication. By extracting these communication patterns from the observation sessions, it was possible to present the knowledge uncovered thus far in a manner that would best benefit novice student pairs. The three patterns are presented next.

4.4.1.1 The Restarting Pattern and guidelines

At several points during the observations, pairs were observed to completely change the topic of discussion from their current work to a more casual topic. For instance, during the *pairwith.us* videos, a member of the pair suddenly interrupts the coding process, and starts talking about his Father's Day plans. Similarly, in a separate observation, the pair starts to discuss a recently released film that they had both watched.

Informal discussions with some of the observed industry pairs indicate that these interruptions are usually conscious ones: whenever a pair was stuck for a period of time,

they would make an effort to break their focus by stopping their current actions and move onto an unrelated topic of discussion.

This is described here as the Restarting Pattern (Figure 38).

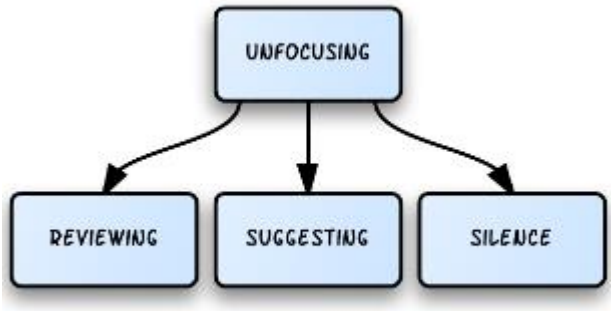


Figure 38: The Restarting Pattern

The data presented in Figure 38 shows that *Unfocusing* is most commonly followed by one of three communication states: reviewing, suggesting and silence. An example of each is given next:

- A reviewing action. The following conversational snippet shows a pair unfocusing (by making jokes about the driver's age), then transitioning into a reviewing state:

D: I've had to turn the font size up. I'm blind.

N: No, you're getting old!

D: I should be wearing glasses, I'm just being stubborn. We had just finished with the casting agent; it was being stubborn.

N: The test is still very much testing the details of the librarian.

- A suggestion. This conversational snippet shows the pair making jokes, with the navigator choosing to bring back focus by making a suggestion for the next stage in their work plan.

D: So what you're saying is 'Terror Wrist'.

N: Yeah, explain the joke. That makes it so much funnier.

D: All my jokes are bad. (laughs)

N: Look at that. You probably want to implement 'help actors get out of character'.

D: Good idea.

- Complete silence. The following conversation shows a pair suddenly unfocusing when the navigator interrupts the coding process. Both programmers have a brief discussion, and then engage in a silent period. This typically ends after the navigator makes a suggestion related to the code.

N: "Don't chop the dinosaur, daddy!"

D: (laughs) Seek help. What's that from?

N: It's from an Australian advert.

D: Right. OK. You keep saying that.

(A period of silence follows.)

Three guidelines suggested by this pattern (Figure 38) are:

- If you and your partner are stuck in a silent period and cannot seem to progress, actively break your focus by discussing something completely off-topic and unrelated to the issues at hand. This will allow you to tackle the problem with a fresh outlook.
- Following this stage, attempt to:
    - Look back on your last couple of steps and review your previous work (review);
    - Identify a fresh start (suggest);
    - Try to think about your end goal when suggesting next steps, in order to make progress (think/be silent).
- If your partner is attempting to break focus, do not dismiss this. Breaking one's focus using jokes and private conversations can lead to a fresh perspective, which you and your partner may need.

4.4.1.2 The Planning Pattern and guidelines

Following a *Suggestion*, a pair was sometimes likely to review the existing code to understand how refining it might help them achieve their main goal. As part of this conversation, one of the pair would typically explain the underlying structure or any legacy code that might be unfamiliar to their partner. This is presented in Figure 39.

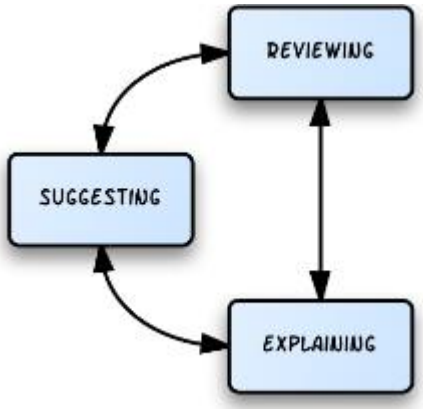


Figure 39: The Planning Pattern

The following conversation illustrates the driver making a suggestion, the navigator reviewing current procedures, and then proceeding on to explaining their reasoning.

> *D: It still feels like we're missing something. We're getting closer to the general solution, though. I'll stick closer to what I have on screen.*
>
> *N: We have* c*, then* b*… and* a *to* b*… and* b *to* c*… to* d*.*
>
> *D: Yep.*
>
> *N: At the moment we're eagerly calling* a *to lock* a *off. If I don't do this, obviously it's going to carry over.*

A suggestion could also separately lead to an explanation – for example, whilst discussing a method, rather than reviewing the code structure, the pair would explain implications that the method would have with respect to their goal. This concept, as well as that of a member asking for clarification by their partner, is also seen as a way to avoid the pair becoming disengaged (Plonka et al., 2012). The following shows a

navigator making a suggestion, and then further explaining how it would impact the written code.

> *N: Could you – double dash. It's one over...*
>
> *D: Yeah?*
>
> *N: It's actually a funny thing. If you whip out an agent test right now, it would generate itself. Because you told it to. Do you get it?*

This pattern (Figure 39) occurred most often at the start of the pairing session: the sessions observed typically started with the pair reviewing legacy code, and attempting to devise ways to reduce error messages or solve problems.

Three guidelines suggested by this pattern are:

- Suggestions and reviews are both useful states that will allow you to drive your work forward. When in these states, feel free to communicate about a range of things; a potential cycle could be as follows:
    - Review previous code
    - Suggest an improvement
    - Review methods to be changed
    - Suggest potential impact
- At any stage, do not hesitate to ask your partner for clarification about any suggestions that they make, or actions they are working on that you do not necessarily understand.

- Think about what your partner is saying and doing. Offering an interpretation of your own understanding of the current state can help move the work forward.

4.4.1.3 The Action Pattern and guidelines

The Action Pattern (Figure 40) occurred mostly whilst a pair was trying to create code. These instances would typically consist of a member of the pair making a suggestion as to what should be coded, or how certain code should be tackled.

The pair would then either talk about the code, or, alternatively, the driver would start muttering. The muttering frequently led to the navigator making suggestions based on what the driver was saying, which acted as a prompt for discussions.

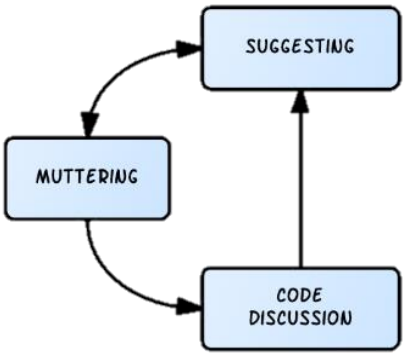


Figure 40: The Action Pattern

The following example shows the navigator suggesting next stages (in this case, to code a certain test). The driver starts muttering. After a while, the navigator interjects, discussing the benefits of the current approach.

N: Excellent. The method's completed. I guess it's time to go on and do the test now.

D: (muttering whilst typing code and running commands)

N: (reacting to the completed method, and the expected results of the test) I think it's a good example of the level of feedback and the cycle time.

Writing code is generally handled by the driver, rather than both members of the pair, thus guidelines arising from this pattern are targeted towards individual members of the pair:

- *(for the driver)*: Whilst you are programming or thinking about how to structure your code, try to be more verbal – for example, by muttering whilst you are typing. This tends to help the navigator to know that you are actively working, and have a clear sense of how you are approaching the task at hand. If you verbalise your thoughts, this will help the navigator make informed suggestions based on your current actions.

- *(for the navigator)*: Whilst the driver is programing, actively look to make suggestions that contribute to the code.

- *(for the navigator)*: If the driver is muttering, use this opportunity to make sure your suggestions have been properly understood.

4.4.2 **The Communication Guidelines**

Figure 41 summarises the communication guidelines extracted from the patterns depicted in Figure 34.

Restarting	If you and your partner are stuck in a silent period and cannot seem to progress, actively break your focus by discussing something completely off-topic and unrelated to the issues at hand. This will allow you to tackle the problem with a fresh outlook.	Following this stage, attempt to: - Look back on your last couple of steps and review your previous work (review); - Identify a fresh start (suggest); - Try to think about your end goal when suggesting next steps, in order to make progress (think/be silent).	If your partner is attempting to break focus, do not dismiss this. Breaking one's focus using jokes and private conversations can lead to a fresh perspective, which you and your partner may need.
Planning	Suggestions and reviews are both useful states that will allow you to drive your work forward. When in these states, feel free to communicate about a range of things; a potential cycle could be as follows: - Review previous code - Suggest an improvement - Review methods to be changed - Suggest potential impact	At any stage, do not hesitate to ask your partner for clarification about any suggestions that they make, or actions they are working on that you do not necessarily understand.	Think about what your partner is saying and doing. Offering an interpretation of your own understanding of the current state can help move the work forward.
Action	*(for the driver)*: Whilst you are programming or thinking about how to structure your code, try to be more verbal – for example, by muttering whilst you are typing. This tends to help the navigator to know that you are actively working, and have a clear sense of how you are approaching the task at hand. If you verbalise your thoughts, this will help the navigator make informed suggestions based on your current actions.	*(for the navigator)*: Whilst the driver is programing, actively look to make suggestions that contribute to the code.	*(for the navigator)*: If the driver is muttering, use this opportunity to make sure your suggestions have been properly understood.

Figure 41: The communication guidelines

4.5 Summary

This chapter introduced studies which confirmed that the coding schemed derived in Chapter 3 was applicable to a set of pairs from two different areas of industry. Observations were carried out with these pairs in their workplace to ensure authenticity of the gathered data. Following coding sessions, an inter-rater reliability test confirmed that the developed coding scheme was suitable to describe and analyse the communication exhibited by various industry-based pairs.

An in-depth analysis of the way the pairs communicated led to the extraction of certain high-frequency transition patterns from the data, which were used in conjunction with the observations to establish guidelines for pair communication.

Chapter 5: Exploratory Study

This chapter describes an exploratory study carried out with a class of undergraduate students across one semester. The purpose was to determine what effects, if any, the application of the guidelines had on the pair programming experience of complete novices.

5.1 Method

The aim of this study was to introduce a group of students to the pair programming guidelines, and to develop an understanding of what effects the guidelines had on the students' experiences of pair programming, compared to their peers. The study was carried out with undergraduate students taking the taught module Agile Software Engineering at the School of Computing, and was structured in two parts, each concentrated around the students' two major assignments. Each part consisted of a number of weekly surveys and a final interview.

5.1.1 Participants: AC31007

One of the taught modules within the School of Computing at the University of Dundee is AC31007: *Agile Software Engineering*, in which third-year students are taught various agile methods. As part of the module, the class was split into teams of 3-5 people for the duration of the semester and were asked to adopt an agile methodology during their two major assignments, both of which were part-completed attempts at tackling a larger software development project.

Rather than grading the students on their code, the lecturer considered how each team of students worked together with respect to several aspects of Agile Methodology; in particular, grades were assigned on a team's project planning and use of source control.

As grades were not assigned based on pair programming or metrics that were deemed useful for this study, grades were not factored into this analysis.

All students were pair programming novices. Prior to the start of the module, none of the students had practised formal pair programming and had no experience of the concept. In a later post-study interview, a team (5 students) said that prior to the module they had helped each other on assignments in a fashion ‘similar’ to pair programming but none of these sessions applied any formal methodology or driver-navigator roles.

The students were asked to attend weekly lectures and 2 hours per week was assigned as lab time. During this lab period, students were asked to always work in pairs within their teams. The teams were assigned together in a random order by the course lecturer, and the students were free to assign pairs within their team, switch roles, and pair rotate as they deemed fit.

5.1.2 **Procedure**

Ethical consent was obtained from the University of Dundee’s School of Computing Ethics Board for all students involved in this study. The 2012 class consisted of 28 students split into 7 teams. All students gave informed consent for observations and interviews to be carried out during the semester for research purposes.

Following a basic introduction to pair programming, Phase 1 was structured as a pre-test period of four weeks during which all students were becoming acquainted with the concept of pair programming and each other, as well as submit their first assignment. All students had the same grounding and understanding of pair programming, which prevented any bias occurring from introducing the guidelines to a subset of the class too early. Following the end of Phase 1, the teams were split into two groups: an

experimental group which would receive a copy of the pair programming guidelines outlined in Chapter 4, and a control group. To make this split seem more natural for the students, the "placebo" guidelines were created for the control group based on 'scrum' since students were frequently practising this technique in the labs. This allowed for the students to view presented guidelines as an additional part of their module: Group A would be studying *Advanced Pair Programming*, and Group B would be studying *Advanced Scrumming*.

Phase 2 consisted of a post-intervention period of another four weeks during which the students prepared for their second assignment.

In both test periods, the researcher handed out optional weekly surveys (see Appendix F) to the students, to track how their perceptions of pair programming and each other's performance as pair partners developed throughout the semester. Furthermore, semi-structured interviews (Appendix F) with members of each team were used at the end of each test period to explore the students' thoughts and experiences on pair programming at these different points during the semester.

5.2 Phase 1

Phase 1 was considered to be the pre-test period of 4 weeks, following each team's first experience with pair programming up until the submission of their first assignment. The researcher was present at each lab session to assist students with any issues. All queries were either technical ones, dealing with the installation and setup of development environments, or module-related ones, dealing with sprint backlogs and assignment completion. All teams did their pair programming outwith the assigned lab times, preferring to use the lab hours as a time to get technical or assignment help. A copy of the survey is available in Appendix F.

5.2.1 **Survey Results**

All surveys used the same 5-point Likert-scale questions about pair programming and communication, asking the students to focus their answers on their experiences during the week. For the questions below (Table 20), each 5-point Likert scale ranged from Strongly Disagree (1) to Strongly Agree (5).

Table 22: Mean and Standard Deviation results from Phase 1 (weeks 1-4)

Question	**Week 1** (11 replies)		**Week 2** (14 replies)		**Week 3** (0 replies)		**Week 4** (1 reply)	
	M	**SD**	**M**	**SD**	**M**	**SD**	**M**	**SD**
This session was enjoyable.	4.1	0.51	3.9	0.26			5.0	-
I feel pair programming is more beneficial than solo programming.	3.5	0.66	3.8	0.67			4.0	-
No periods of uncomfortable silence.	4.4	0.64	4.1	0.83			5.0	-
I found communicating to be easy.	4.5	0.66	4.2	0.77			5.0	-
I was confident.	4.2	0.72	3.6	0.61			5.0	-
My partner contributed during this session.	4.2	1.11	4.2	0.77			5.0	-

There were no replies in week 3 and only one student replied in week 4. This was due to the fact that most students did not attend the scheduled lab session, but chose to work at times other than the regularly scheduled periods when the surveys were distributed. During the latter stage of Phase 1, students were also concerned with finalising their assignment (due during week 4), and therefore may have preferred to focus on that, rather than on completing optional surveys.

5.2.2 **Survey Discussion**

Due to the anonymity of the surveys, it is not possible to make statistical comparisons between means in weeks 1 and 2 which match with the students' first experiences of pair programming in a lab environment.

The results show minor increases in the mean score for *pair programming is more beneficial than solo programming*. Whilst the data does not cover a period of time long enough to draw conclusions, it does indicate that by the second week, the students felt that a pair programming approach was beneficial, and that their pair partner was making more contributions.

The mean scores for questions dealing with communication, confidence, and enjoyment of the session showed slight decreases between weeks 1 and 2. Once again, the data does not range over a period of time substantial enough to draw conclusions. However, it may be that despite finding the pair model beneficial, students were dealing with anxiety with their pending assignment deadline, as well as their pair work in particular, mirroring issues seen in other papers that discuss communication being a barrier to pair programming for novices (Williams et al., 2000, VanDeGrift, 2004).

5.2.3 **Interview**

5.2.3.1 Procedure

Following the conclusion of Phase 1, an interview was carried out with each team to understand their initial perceptions and experiences of pair programming. The interviews were semi-structured (Appendix F), allowing the base skeletal structure to be adjusted by the interviewees' responses (Robson, 2011).

Each half-hour interview started with the researcher introducing the aim and also explaining that all opinions and answers would be kept anonymous. Any module grades could not be affected by opinions expressed during the interview. The interviews focused on the team's pair programming perceptions and experience, ending with a conversation on scrumming practices. A discussion about the former is presented next; a discussion on the latter falls outside the focus of this thesis and is therefore not included.

Each interview was captured on a voice recorder, then immediately transcribed following the session.

5.2.3.2 Results

When asked about their expectations of pair programming (prior to having tried it out), students admitted to being "apprehensive, and very, very nervous" at the prospect of working closely with a partner, comparing the concept to "a stupid idea" and "a waste of effort". One student in particular assumed that "since [we got shown it], it must be useful". However, all teams felt initially that the experience would be negative.

Once they put pair programming in practice, however, these opinions changed. One student commented, “You don’t really set out to meet particular goals, but it somehow ends up seeming to work out a lot better”. Students agreed that generally, having a second set of eyes helped “keep up morale”, and that after using it, they “could see the merit”. Despite their initial negative impression, “it was quite productive”. It was pointed out that “it takes time to get into the practice” and it was “all about levelling ourselves”, and that despite the benefits, initially “you could make more mistakes because you’re nervous”. One team described the constant need to explain themselves: “We had to get fluent with our experience [and ask ourselves] *why am I typing a certain thing?* [as] we had to explain it.”

The general reaction was that the students found that their implementation of pair programming was “impractical” due to various timetabling issues. Each student within the team had different commitments: “You can’t really practically do it to its fullest potential because of all the other modules but… I feel it could work better if we had a whole day, in a real-world environment.”

Overall, the class had mixed reactions to the communication aspect of pair programming. Whereas some teams felt they “gelled well” and that they “all got along”, some other teams said that “it was more bickering; *I want to do it like this,* or *I want to do it like that*”. One of the other teams classed this as “micro-arguments; on again, off again. We had a lot of them”, explaining that “it can be quite embarrassing completely pointing out someone’s mistakes”.

The ratio of students who preferred to drive against students who preferred to navigate was 55:45, which is close to the 60:40 ratio indicated by Bryant et al. (2006).

Interviewed students were divided between their preference to drive or navigate, which some students simply saying that “either one was good”. Most students, however, spoke about a distinct preference for one of the roles.

Students who preferred the navigator role stated that it was “much easier to fix problems than to create them”, with students feeling that they felt “more capable of doing the logic than actually getting the syntax correct”. Some navigators felt that they were in a position of more control over the driver (“when the driver is stalled, the navigator takes some time to get into the rhythm – but can ultimately solve things”). One student always chose to act as navigator in his pair, as they “did not like the whole ‘someone looking over my shoulder’ idea”.

Drivers, on the other hand, largely chose to do so because of their affinity with the code. As previous ‘solo’ programmers, several students indicated that they felt more comfortable in this role: “normally I prefer to be coding anyway”/“I didn’t prefer one role over the other, though I generally ended up in the Driver role”. A group of students mused that “driving is stressful – but this is one of the reasons [pair programming] has such good results: you’re constantly focused on the code”.

Several students spoke about experiencing both roles, and understanding the benefits of both: “I thought navigating would be really boring – but I ended up seeing the merit of it”. One team assigned roles based on their understanding of the IDE and the language used: “when I understand the language I like to Drive – but when I’m trying to learn, I’d rather Navigate”.

The Phase 1 interview raised relevant issues such as timetabling, which teams agreed they needed to focus more energy on. There were mixed reactions to the communication

question, with some teams finding their intra-pair communication to be straightforward, whereas others found it to be awkward and prone to arguments.

5.3 Phase 1 to Phase 2

For Phase 2, the teams were randomly split into two groups: Group A (consisting of 16 students), and Group B (consisting of 12 students), for the delivery of the guidelines, with the latter acting as a control group. The students were given two separate hour-long lectures during which they were told that this was part of their focus for the rest of the semester: Group A would be practising *Advanced Pair Programming*, and Group B would be practising *Advanced Scrumming*. Each lecture focused upon guidelines: Group A was given the pair programming guidelines described at the end of Chapter 4, whereas the module co-ordinator created scum guidelines that were provided to the control group (B). Teams within each group were then asked to use the guidelines as and when necessary for the duration of the semester.

5.4 Phase 2

The Phase 2 (post-intervention) period consisted of another four weeks during which students were working on their second assignment of the semester. Students were once again surveyed.

5.4.1 Survey Results

The survey was applied as per Phase 1. The results are presented in Table 23.

The response rates were poorer than those reported in section 5.2 above. Students rarely attended the time-tabled lab session, and even if they were handed a survey, they seldom returned it. By this point, the semester was reaching its end, and students cited multiple assignments and pending exams as being too important and time-consuming for them to remember to fill in weekly surveys.

Table 23: Mean and Standard Deviation results from Phase 2 (weeks 6-9)

Question	**Week 6** (4 replies)		**Week 7** (0 replies)		**Week 8** (3 replies)		**Week 9** (3 replies)	
	M	**SD**	**M**	**SD**	**M**	**SD**	**M**	**SD**
This session was enjoyable.	4.0	1.00			4.0	0.00	4.0	0.00
I feel pair programming is more beneficial than solo programming.	4.3	0.83			4.0	0.00	4.3	0.47
No periods of uncomfortable silence.	4.0	1.22			3.7	0.47	4.3	0.47
I found communicating to be easy.	4.0	1.00			4.3	0.47	4.0	0.00
I was confident.	3.5	1.50			3.3	0.94	4.0	0.00
My partner contributed during this session.	4.0	0.00			4.0	0.82	4.3	0.47

Nonetheless, for the duration of the surveyed time, it can be seen that a number of the reported attitudes were quite positive: the students were enjoying the pair programming experience, and felt that pair programming was more beneficial. Students also found communication to be relatively easy.

5.4.2 Survey Discussion

At this stage, it is important to consider the fact that the survey reported in Table 23 consists of scores provided by students from both Group A and Group B. Some items on the survey, such as communication, partner contribution, and enjoyment, would benefit from being analysed against each respective group to understand whether the pair programming guidelines had an impact on these reported items for the exposed group, as opposed to the control group. A larger response set would have allowed comparisons to be made between the exposed group and control group responses.

5.4.3 Interview

5.4.3.1 Procedure

Following the conclusion of Phase 2, an interview was carried out with each team to understand their perceptions and experiences of pair programming following the exposure period. Semi-structured interviews were used as the conditions discussed in section 5.2.3 were still relevant for this phase of the study.

Following the half-hour interview, each team were debriefed from the study as per the conditions of ethical approval; therefore, all students from the control group who attended the interview were exposed to the pair programming guidelines for the first time. Additional informal feedback was collected from students who had just been exposed to the pair programming guidelines. All students had been actively pair programming for ten weeks.

Each interview was captured on a voice recorder, then immediately transcribed following the session. For the purposes of this interview, a single A4 sheet consisting of

the pair programming guidelines was provided to all teams as a discussion prompter after an initial round of information gathering.

5.4.3.2 Results

When asked whether their expectations of pair programming matched the actual experience, all teams in both groups agreed that they could understand "where and why it [is] useful", admitting that they felt that the pair programming process was more natural than it had seemed at the start of the semester. None of the teams indicated that communication was an issue during this phase.

All students in Group A teams indicated that they found the pair programming guidelines to be beneficial, as evidenced by the following quotes:

> *"I found that the restarting pattern came in useful when I was thinking about other modules as well… the action pattern, and noticing the driver muttering, was useful."* – Team 1

> *"The [restarting guideline] would be the most useful one, whereas [planning and action] would come more naturally. They are definitely good if you don't know your partner well."* – Team 3

> *"They seem like really good tips if you get stuck; a lot is self-explanatory, which is good."* – Team 3

> *"I think we definitely used the restarting pattern. [You] definitely pick up on when people are getting frustrated, so we went out to the shop; getting away from the computer was helpful."* – Team 5

When asked their opinion regarding introducing the guidelines as a taught component that complemented an introduction to pair programming, there was disagreement between teams. Some students argued that pair programming should be fully understood prior to the introduction of the guidelines: "it was good to get to grips with pair programming [by themselves], and learn from [their] mistakes before being taught [the guidelines]", and "it might have been too much information at the start". Conversely, Team 1 felt that the concepts could have been introduced earlier:

> *"At the start of the course there was a lot of repetition, whereas the concept is very straightforward: you are in a pair, and programming. Being given these guidelines would have shown the more advanced side at the start, I think".* – Team 1

Teams in Group A agreed that following the initial lecture at the end of Phase 1, the guidelines were not something they needed to actively think about in order to implement:

> *"We did a lot of it without thinking about it."* – Team 4

> *"We followed them because they occurred naturally."* – Team 5

These comments are encouraging, indicating that the guidelines were adopted quite naturally by the student teams. Teams found them to be useful in different situations and scenarios than those initially envisioned by the researcher. For example, one student spoke about how her pair used the planning guidelines to learn and understand how to write Android code from scratch.

The control group were all presented with a copy of the guidelines during the debrief session, but (as with the experimental group) were not told that they were the product of the interviewer's research, but rather that they were advanced tips on how to collaborate effectively within their pair.

The reactions from the control group were highly positive. All teams recognised the guidelines as patterns of interaction that they had followed:

> *"The whole breaking focus thing... seemed to help for ours. It's the whole 'you find all your ideas in the shower' thing, where you don't think about it – and it comes to you."* – Team 2

> *"The restarting one definitely looks like something we did [..] – I believe we did all three patterns."* – Team 6

> *"Just looking at it, we did tend to fall into the [planning] one – we did a small amount with suggestions, and if there was a disagreement we would explain and try to come to a consensus; I think we fell into a similar idea to that."* – Team 7

Furthermore, all teams discussed potential benefits of having early exposure to the guidelines:

> *"There's a definite benefit in introducing this. In pair programming we're told to 'work in pairs: go!', and there weren't formal steps, apart from the fundamentals. There wasn't a lot of what to do if you became stuck".* – Team 2

> *"Not so sure if these were to be presented [...] to help pair programming, as they are pretty straight forward – and will be done, most likely."* – Team 6

> *"It might have helped in the start."* – Team 7

There was consensus that the guidelines are beneficial, but similarly to Group A, there was no consensus about whether the guidelines should be introduced early on in a student's pair programming learning or later on. The comments made by Team 6 in particular were interesting – the team debated whether the guidelines should be presented to novice pairs early on, but decided against it, as they felt that novice pairs would 'most likely' discover them in due course.

5.5 Limitations

Despite initial enthusiasm from all students, a limited number of people completed the weekly surveys, leading to data which could not be treated as indicative. It had been expected that this data would be obtained during their scheduled lab hours. However, students were permitted to use the lab at any time of day and so surveys were not completed at these set times. Discussed during interviews, students largely reported that they had been too busy to complete a weekly survey which – in their eyes – was not immediately valuable to them.

As students submitted their surveys in batches, the low response rate was only evident at the conclusion of the study. It is clear that an alternative arrangement might have been more successful at collecting continuous data, such as moving the surveys to an online-based system (with e-mail or text reminders) or providing the survey at lecture times rather than lab times. Nonetheless, the results obtained still provide a valuable insight into the students' perceptions of pair programming, and the associated guidelines.

5.6 Summary

The pair programming guidelines developed in earlier chapters were presented to novice pairs for the first time, with the aim of getting student feedback throughout the course of a three-month semester.

Following final interviews, it appears likely that the communication guidelines were viewed as beneficial and useful by novice-level pair programmers – but it is not clear what effect, if any, they had on the pair, and on the individual developers within the pair. Some teams from both groups seemed to indicate that the guidelines would be more beneficial for pairs in which the individuals are not used to talking to each other; "maybe if you had not spoken to [your partner] before, you might be hesitant to ask questions. [They are] definitely good if you didn't know your team, or partner, well."

This is a positive initial result, as it shows that novice pairs reacted positively to the guidelines and that they are seen as natural and potentially beneficial. However there is a need to understand what effects the guidelines have on novice pair behaviour, and to what extent they alter communication behaviours.

Chapter 6: Evaluations of the Guidelines

This chapter describes three studies that were carried out with several novice student pairs to investigate their experience using the pair programming guidelines. Each of these studies report the student experience on two measures: ease of communication and perceived partner contribution. The chapter concludes with a discussion of the results obtained, and implications that these have on the current research.

6.1 Aim of the Studies

The research question introduced in the literature review is: *Can extracted communication patterns from expert pair programmers be used to help novice student pairs to improve their intra-pair communication?*

The aims of this chapter are tied with the latter part of the question: can the guidelines cast from the patterns be used to help novice student pairs to improve their intra-pair communication? The qualitative work carried out in Chapter 5 suggested that students are willing to use the guidelines, and that the guidelines could have certain benefits. Quantitative results would give added understanding of the effects that guidelines have on novice students. To that end, a series of studies have been planned to understand whether the guidelines can positively impact the students' experience of communication.

In the literature review (Chapter 2), it was seen that 'communication' is often seen as a barrier to successful pair programming for first-time pairs (Williams et al., 2000, Begel and Nagappan, 2008, Sanders, 2002). Furthermore, unequal participation is one of the top perceived problems for students (Srikanth et al., 2004, VanDeGrift, 2004). As the guidelines have been developed to improve this communication, the studies have been

designed to investigate these two issues, with pairs reporting on their experience of communication, particularly with respect to how easily they were able to communicate with their partner (referred to as ‘ease of communication’) and on their partner’s contribution to the pairing session (‘perceived partner contribution’).

6.2 Method

Each study is set up in a similar manner, with a number of pairs working through several tasks. The first two studies (“Parts 1A & 1B”) present tasks that involved code reviewing and debugging. The final study (“Part 2”) involves programming tasks.

In each case, pairs of students were recruited and randomly allocated to one of two groups: a test group, which would be exposed to the guidelines prior to the set task, and a control group. Each pair was asked to complete as many tasks as possible during a 45-minute time limit. This was followed by a post-test survey, during which individual members of each pair rated their experience.

In order to extract conclusions with relevance to the effect of the guidelines on novice pair programming communication, the following measures were taken in each study:

- *Ease of Communication,* measured by looking at the individual post-study Likert scales;
- *Perceived Partner Contribution,* measured from the individual post-study Likert scales.

For each of the studies, the Likert scale data resulting from the post-test surveys were analysed to determine whether there were any significant statistical differences reported between the students who were exposed to the guidelines and those who were not.

Student success (in terms of correct solutions) was also measured and is discussed in each section below.

As the data used in these analyses is extracted from Likert scales (and therefore 'ordinal'), the Mann-Whitney U test was applied (Ryu and Agresti, 2008). Furthermore, as a non-parametric test, this is more robust against certain assumptions (e.g. outliers seen in the data) (McElduff et al., 2010).

The following null hypotheses are tested in each study:

1. H_0: The distribution of the pair's ease of communication is equal across the two groups.

   H_A: The distribution of the pair's ease of communication differs by exposure to the guidelines.

2. H_0: The distribution of the pair's perceived partner contribution is equal across the two groups.

   H_A: The distribution of the pair's perceived partner contribution differs by exposure to the guidelines.

3. H_0: The mean number of completed tasks for pairs who were exposed to the guidelines and pairs who were not exposed is equal in the population.

   H_A: The mean number of completed tasks for pairs who were exposed to the guidelines and pairs who were not exposed is not equal in the population.

6.3 Procedure

Each of the studies follows a similar procedure, described in this section:

Ethical approval was obtained from the University of Dundee's School of Computing Ethics Board for all participants involved in the studies described in this chapter.

An e-mail was circulated to undergraduate students reading for a Computing degree, inviting them to participate. All participants had previous experience of using Java as a programming language. Pairs were randomly set up so that each pair consisted of students at the same level of study. As much as possible, within each level, 50% of the pairs were randomly allocated to a group which would be exposed to the guidelines, leaving the rest of the sample as a control group.

Pairs were separately invited to a test room. If they had been randomly assigned to the experimental group, the pair was first exposed to the guidelines through the use of the prepared video, paper guidelines, and a verbal presentation by the researcher.

A camera and a voice recorder were set up in the test room to allow for data capture. Each pair was provided with a laptop consisting of the task they were required to solve (which differed between the studies reported in Parts 1A and 1B, and the study reported in Part 2). The pairs were each given 45 minutes to sequentially work their way through as many tasks as they could. The recording devices were then switched on and the researcher left the room.

Following the test period, the researcher returned, logged the number of programs attempted and distributed post-study surveys (Parts 1A/1B: Appendix G; Part 2: Appendix I) to be completed by the individual members of the pair.

In each of the surveys, two questions queried the individual on their experience with development as a solo programmer, and as a pair programmer. This data was used to

measure central tendencies and variance within the groups, in order to ascertain if there were any significant differences between the groups that could bias the results.

The remaining questions asked the individual to rate their perception of benefit of pair programming over traditional programming, the ease of communication during the session, and to rate their partner's contribution.

In the Part 2 survey of the study, students were also asked to note which role they had assumed (i.e. driver or navigator) during the recorded session. The resulting data was used to inform the discussion reported in section 6.5.4 below.

The pair's code was then reviewed by the researcher and the number of successfully completed programs was recorded. This was used to understand whether the guidelines had any significant impact on the pair's success rate.

6.4 Part 1: Code Review and Debugging Studies

In 2010, Murphy et al. published a paper discussing conversations within pairs, focusing particularly on statements related to a series of tasks that the pair were asked to debug, with the aim of gaining a better understanding of how pairs work together to find and fix bugs, through the analysis of their verbal communication. The study was carried out with ten undergraduate students, and used a set of 19 Java programs with logical errors (Appendix H) as the code-base for this task. All programs given to the students would compile, but would not display the correct output. Pairs were given 45-minutes to go through the list of programs and solve as many of the logical errors they could. The researchers then explored the students' verbal interactions in order to extract general observations of the pair's discourse whilst carrying out these debugging tasks. The

authors note that they “found that pairs that talked more […] attempted to solve more problems” (Murphy et al., 2010).

The studies described in this chapter explore whether exposure to the pair programming guidelines can affect the way pairs perceive their experienced communication. To present pairs with tasks that would generate discussion, the code-base written and used by Murphy et al. was presented as the main task for both Parts 1A and 1B. It is important to note that the studies reported here are not replications of the Murphy et al. study, but simply make use of the same materials. The authors were contacted, and gave permission for the use of the buggy programs.

6.4.1 **Part 1A: Code Review Study**

The aim of this study was to understand whether the pair programming guidelines could have an impact on the students’ communication experience whilst they were pair programming.

6.4.1.1 Study Design

The study was carried out during a two-week period, following the method described in section 6.2.

All pairs (n=13) were given a list of the first nine buggy programs and a laptop with a copy of the code. The buggy code used consists of 19 programs (Appendix H), each of which has one logical error. For example, program #1 takes in three numbers and calculates a mean average: however, due to the use of the wrong variable types, it generates the wrong answer (for example, the average of ‘2, 2, 3’ is given as ‘2.0’,

rather than ‘2.3’). Each program was tested using the NetBeans IDE, and compiles successfully with no syntax errors or warnings.

The pairs were informed that each program contained one logical error, and that they had to solve as many errors as possible within the 45-minute time limit specified.

The use of code review helps programmers “identify a majority of program defects”, especially at the more novice level (Chmiel and Loui, 2004), as programmers are focused on spotting and fixing errors without relying on a compiler. This study, all participants were asked to fix the code by hand, as it was presumed that use of a compiler would have revealed the original bug without any need for the participants to debate and discuss potential solutions.

6.4.1.2 Participants

The following numbers of students were recruited:

Further Education College: 6 students (3 pairs)

Level 1 (undergraduate): 12 students (6 pairs)

Level 3 (undergraduate): 8 students (4 pairs)

Pairs were set up so that each pair consisted of students at the same level of study (year group).

At each level, 50% of the pairs were randomly allocated to the group which would be exposed to the guidelines (n=7 pairs), leaving the rest of the sample (n=6 pairs) as a control group. (In the instance where the pairs could not be evenly split between the experimental and control groups, an extra pair was placed in the former group.)

The next section presents a detailed discussion analysing the participants' previous experience with both solo programming and pair programming. This will be followed by a more detailed description of the study and the post-test reports, as well as an analysis on the participants' reported experience with communication for this study session.

6.4.1.3 Participant Experience

Previous Programming Experience

The students' reported experience with solo and pair programming was analysed to determine if there were any statistically significant differences between the two groups which may otherwise affect the data.

Table 24: Student programming experience

	Exposed		**Not Exposed**	
	M	**SD**	**M**	**SD**
Solo Programming Experience (years)	3.7	3.90	2.4	1.69
Pair Programming Experience (years)	0.2	0.22	0.6	1.21
Previous Pair Programming Experience with this Session's Partner (years)	0.0	0.09	0.0	0.00

The SD is greater than the mean in some of these cases due to outliers within the data; for example, whereas most students put down solo experience of 1-4 years, two students had solo experience of 10 and 13 years respectively. The data was therefore analysed using the Mann-Whitney U tests: as a non-parametric test, it is less likely than the t-test to be affected by assumptions not holding (e.g. outliers in the data) (Field, 2009). This test was hence used to analyse the data.

The data show that the groups had somewhat different levels of experience; on average, more individuals in the "exposed" pairs had solo programming experience, but more individuals in the "non-exposed" pairs had pair programming experience. Furthermore, two pairs within the exposed group had previous experience in pairing together. Statistical tests were carried out to establish whether the differences between the two groups were significant and whether they might cause the results to be biased:

- No significant differences in 'solo' programming experience were found between the experimental and control groups: $U = 69.5$, $z = -0.414$, $p = 0.687$ ($p > 0.05$).
- Similarly, no significant differences in pair programming experience were found between the experimental and control groups: $U = 85$, $z = 0.056$, $p = 1$ ($p > 0.05$).
- Only one pair reported any previous experience with their partner. No significant differences were found between the experimental and control groups in 'previous pair experience with today's partner': $U = 72$, $z = -1.336$, $p = 0.181$ ($p > 0.05$).

The results show that there were no significant differences between the two groups, and that further analysis should not be skewed by any bias resulting from one group having additional previous experience.

Perceived Benefits of Pair Programming

As part of the post-test survey, students were asked to rate the statement '*I feel pair programming is more beneficial than solo programming'* on a 5-point Likert scale, ranging from 1 ("Strongly Disagree") to 5 ("Strongly Agree").

Figure 42 charts student responses between the two groups:

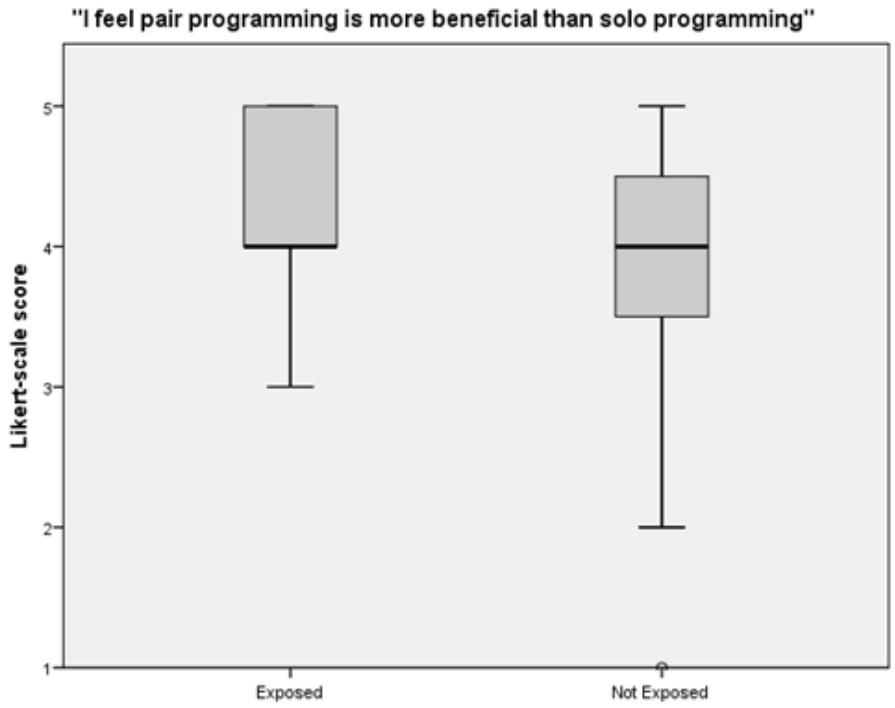


Figure 42: Reported scores for "I feel pair programming is more beneficial than solo programming".

It can be seen that the exposed group (M=4.4, SD=0.63) show less variability in their reported answers, whereas the group of students who were not exposed (M=3.8, SD=1.22) report a lower mean and a higher variability. However there was no significant difference in perceived pair programming benefit between exposed students (*Mdn =4*.0) and unexposed students (*Mdn* = 4.0), $U = 60.5$, $z = -1.323$, $p = 0.186$.

These results show that following the session, the student perception was that pair programming was more beneficial than solo programming, irrespective of whether they were exposed to the guidelines or not.

6.4.1.4 Results

The Likert scale data resulting from the post-test surveys (Appendix G) were analysed to determine whether there were any significant statistical differences reported between the students who were exposed to the guidelines and those who were not.

As each individual completed their own post-test survey, the population consisted of 26 students: 14 of whom were exposed and 12 students who were not.

The tests that follow compare data between the groups for the following Likert scale items:

- Ease of Communication, reported through the statement, "*During this session, I found communicating with my partner to be easy*".
- Perceived Partner Contribution, reported through the statement, "*Rate your partner's contribution to today's session"*.

Shapiro-Wilk tests were carried out to understand whether the data being analysed were normally distributed. *Ease of Communication* scores for both exposed and unexposed groups were not normally distributed ($p < 0.05$). Similarly, scores for *Perceived Partner Contribution* for both groups were not normally distributed ($p < 0.05$). As the data are not normally distributed for both sets of scores, non-parametric tests were used.

Ease of Communication

Table 25: Descriptive Statistics for Ease of Communication (Part 1A)

	Exposed		**Not Exposed**	
	M	**SD**	**M**	**SD**
Ease of Communication	4.6	0.51	3.9	0.90

Figure 43 depicts the distribution of scores reported by students for *ease of communication* (ranging from 1 ("strongly disagree") to 5 ("strongly agree")) between the two groups. It can be seen that the students who were exposed to the guidelines reported a greater ease of communication than students who were not.

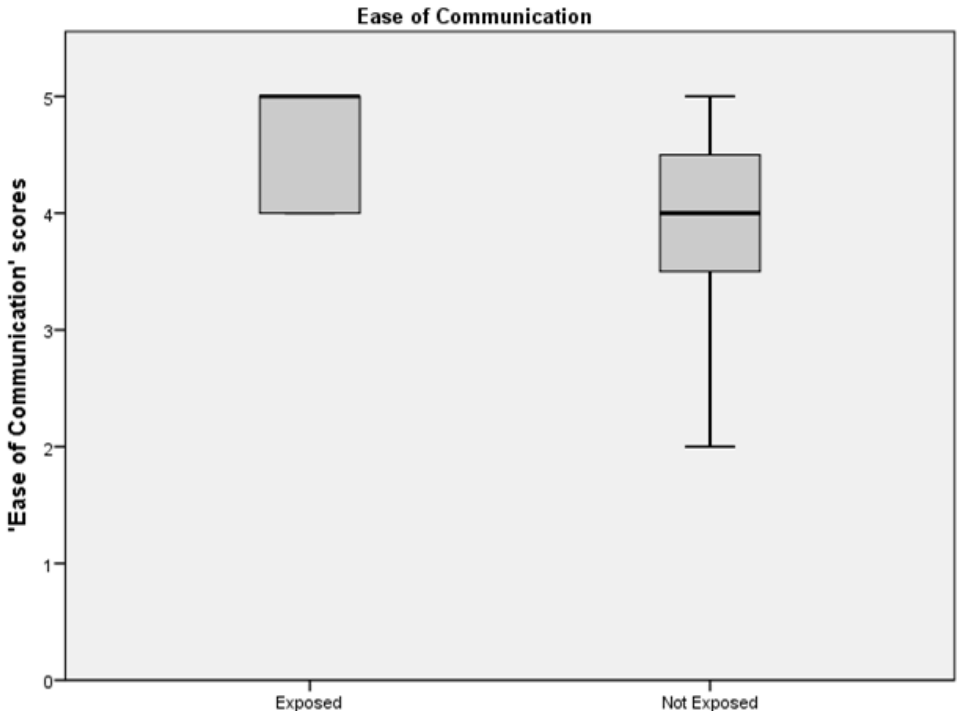


Figure 43: Reported scores for ease of communication

A Mann-Whitney U test was run to determine if there was a significant difference in Ease of Communication between the exposed and unexposed groups. There was a statistically significant difference in ease of communication scores between exposed students (*Mdn* = 5.0) and unexposed students (*Mdn* = 4.0), *U* = 48, *z* = -2.037, *p* = 0.042. In this case, $p < 0.05$, therefore the null hypothesis was rejected.

Perceived Partner Contribution

Table 26: Descriptive Statistics for Perceived Partner Contribution (Part 1A)

	Exposed		**Not Exposed**	
	M	**SD**	**M**	**SD**
Perceived Partner Contribution	4.8	0.426	4.2	0.835

Figure 44 shows the distribution of Likert scale scores for students' *perceived partner contribution* (ranging from 1 ("no participation") to 5 ("excellent")) between the two groups. It can be seen that students who were exposed to the guidelines rate their partner's contribution to be quite high, with relatively low variance.

The asterisk indicates outliers in the data – three of the exposed students reported their perceived partner contribution to be '4'.

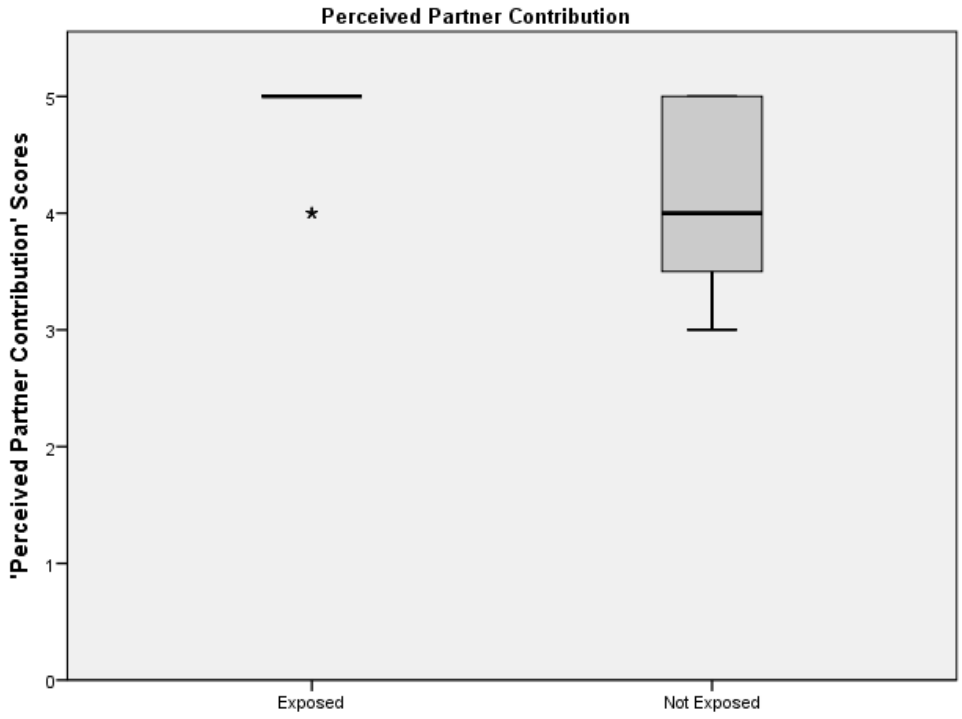


Figure 44: Reported scores for perceived partner contribution

A Mann-Whitney U test was run to determine if there was a significant difference in Perceived Partner Contribution between the exposed and unexposed groups. There was a statistically significant difference in perceived partner contribution scores between

exposed students (*Mdn* = 5.0) and unexposed students (*Mdn* = 4.0), *U* = 48.5, *z* = -2.113, *p* = 0.035.

In this case, p < 0.05, therefore the null hypothesis was rejected.

Successfully Completed Programs

An independent-samples t-test was run to determine if there were differences in completion scores between pairs who were exposed to the pair programming guidelines (n=7), and those who were not (n=6).

There were no outliers in the data, as assessed by inspection of a boxplot (below). The tasks completed for each level of exposure were normally distributed, as assessed by Shapiro-Wilks test (p > 0.05), and there was homogeneity of variances, as assessed by Levene's Test for Equality of Variances (p = 0.385).

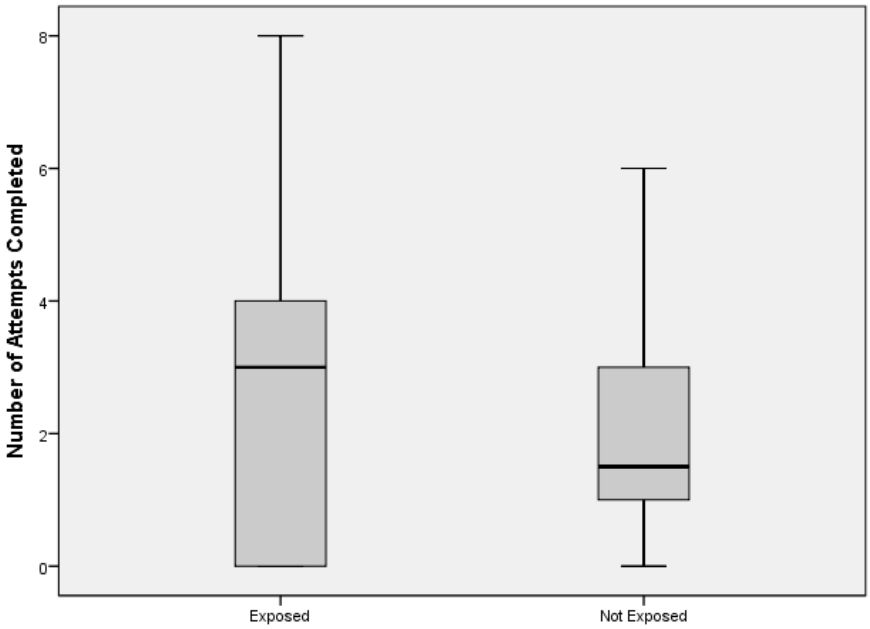


Figure 45: Number of tasks completed in Part 1A

The exposed pairs completed a slightly greater number of tasks completed (2.71 ± 3.04) than the unexposed pairs (2.17 ± 2.14). The difference is not statistically significant; $t(11) = 0.369$, $p = 0.718$.

As $p > 0.05$, the null hypothesis is not rejected.

6.4.1.5 Comparisons between Study Levels

The results reported in sections 6.4.1 and 6.4.2 above indicate that when considering the whole group of observed students, significant differences were reported for *ease of communication* and *perceived partner contribution*. This indicates that students who were exposed to the guidelines experienced improved communication and partner contribution during the study.

Since the observed students belonged to three different levels (year groups) (college-level, undergraduate year 1, undergraduate year 3), further analysis was performed to understand which, if any, levels of study reported the most benefit from the guidelines. As previous analyses indicate that exposed students rated higher communication scores, it was decided to explore any differences that emerged between the exposed students in the various year groups. Therefore, for the purpose of this analysis, only data obtained from the exposed students was considered.

Due to the fact that self-reported Likert data was analysed (for both *ease of communication* and *perceived partner contribution*), the non-parametric Kruskal-Wallis test was used to determine if there were differences in reported scores between student level groups who had been exposed to the guidelines. This test evaluates whether there

are any statistically significant differences between the distributions of three or more independent groups (Field, 2009). The results of this test are reported below:

- For *ease of communication*, there was a slight change in reported scores across study levels (college students: *Mdn* = 4.5; level 1: Mdn = 4.0; level 3: Mdn = 4.0), but the differences were not statistically significant, $\chi 2(2) = 4.153$, $p = 0.125$.

- For *perceived partner contribution*, There was no change in reported scores across study levels (college students: *Mdn* = 5.0; level 1: *Mdn* = 5.0; level 3: *Mdn* = 5.0), and the differences were not statistically significant, $\chi 2(2) = 1.510$, $p = 0.470$.

6.4.1.6 Limitations and Discussion

The aim of this study was to understand whether the pair programming guidelines could have an impact on the students' communication experience whilst they were pair programming. It was initially expected that not allowing the use of compilers would promote further discussion within the pair – however, there is no literature to support this. The participants were arguably not pair programming, but 'code reviewing': a process that may not have been as natural to some of the participants as the usual compiler-aided debugging. Neither of the hypotheses can be accepted as a result of this study. This evidence shows that the guidelines were perceived to improve the participants' intra-pair communication skills in a code review setting. Further studies are required in order to ascertain the validity of these results in a programming context:

1. Students who were exposed to the guidelines reported significantly higher scores for ‘ease of communication’.

2. Students who were exposed to the guidelines reported significantly higher scores for ‘perceived partner contribution’.

3. There is no significant difference in successfully completed programs between students who were exposed to the guidelines, and students who were not exposed.

When comparing the students’ self-reported scores, results show that no particular year group performs significantly differently – exposure to the guidelines does not immediately benefit one year group over the other. This suggests that the guidelines are worthwhile across all study levels observed, and that their use is not solely restricted to completely novice (i.e. level 1) students, but that more experienced students (i.e. level 3) can also benefit from them.

This further suggests that there might be a wider audience for the guidelines beyond what has already been observed beyond the current scope, e.g. with experienced developers in industry who are just starting to pair program. These points will be further discussed in the Future Work section of Chapter 8.

6.4.2 **Part 1B: Debugging Study**

Whilst Part 1A of the study has some initially promising results, these are limited by the fact that students did not have access to a compiler for the duration of the study, and thus were engaged in reviewing code in a way that may have been unnatural to them.

The aim of Part 1B is to understand whether similar results are obtained when students are allowed to use a compiler.

6.4.2.1 Study Design

The design for Part 1B of the study is similar to that of Part 1A, as reported in section 6.2. Pairs who were part of the exposed group were re-exposed to the guidelines prior to the study through a video, paper guidelines, and a verbal presentation by the researcher.

As students had completed up to program #10 during Part 1A of the study, they were asked to attempt programs #11 to #19 from Murphy et al.'s (2010) study for Part 1B. This prevented pairs from having any familiarity with the code that may have skewed or otherwise have affected the results.

All pairs were given a list of these buggy programs and a laptop with a copy of the code. The pairs were informed that each piece of code consisted of a logical error, and that they had 45 minutes to sequentially fix as many programs as they could. All programs given to the students would compile, but would not display the correct output.

Due to the smaller number of participants (and thus, year group distribution) comparisons between study levels (as reported in section 6.4.1.4) would not have yielded any significant results and therefore, this analysis was not repeated.

6.4.2.2 Participants

All participants from Part 1A were invited to this second part via e-mail, which was scheduled four weeks after the first study. In an attempt to replicate as much of the original attempt as possible, each participant was invited to participate with their partner from Part 1A.

A total of ten participants (five pairs) applied for Part 1B, and were separately invited to the study room. The pairs were placed into the same groups as the original study, leading to an exposed group (n=3 pairs) and an unexposed control group (n=2 pairs).

The next section presents a detailed discussion analysing the participants' previous experience with both solo programming, and pair programming. This will be followed by a more detailed description of the study and the post-test reports, as well as an analysis of the participants' reported experience with communication for this study session.

6.4.2.3 Participant Experience

Previous Programming Experience

The post-test surveys were analysed first. The students' experience with programming and pair programming was analysed (Table 27) to ensure that there are no statistically significant differences between the two groups which may have otherwise affected the data.

Table 27: Student programming experience

	Exposed		Not Exposed	
	M	**SD**	**M**	**SD**
Solo Programming Experience (years)	6.0	4.85	1.8	1.61
Pair Programming Experience (years)	0.3	0.22	1.1	1.93
Previous Pair Programming Experience with this Session's Partner (years)	0.1	0.13	0.0	0.00

As before, a Mann-Whitney U test was used to analyse the data.

The data shows that the groups had somewhat different levels of experience; on average, more individuals in the "exposed" pairs had solo programming experience, but more individuals in the "non-exposed" pairs had pair programming experience. Furthermore, two pairs within the exposed group had previous experience in pairing together. Statistical tests were carried out to establish whether the differences between the two groups were significant and whether they might cause the results to be biased:

- No significant differences in 'solo' programming experience were found between the experimental and control groups: $U = 5$, $z = -1.492$, $p = 0.171$ ($p > 0.05$).

- Similarly, no significant differences in pair programming experience were found between the experimental and control groups: $U = 13$, $z = 0.224$, $p = 1$ ($p > 0.05$).

- Only one pair reported any previous experience with their partner. No significant differences were found between the experimental and control groups in 'previous pair experience with today's partner': $U = 8$, $z = -1.225$, $p = 0.476$ ($p > 0.05$).

The results show that there were no significant differences between the two groups, and that further analysis should not be skewed by any bias resulting from one group having participants with additional previous experience.

Perceived Benefits of Pair Programming

As part of the post-test survey, students were asked to rate the statement '*I feel pair programming is more beneficial than solo programming'* on a 5-point Likert scale.

Figure 46 charts student responses between the two groups:

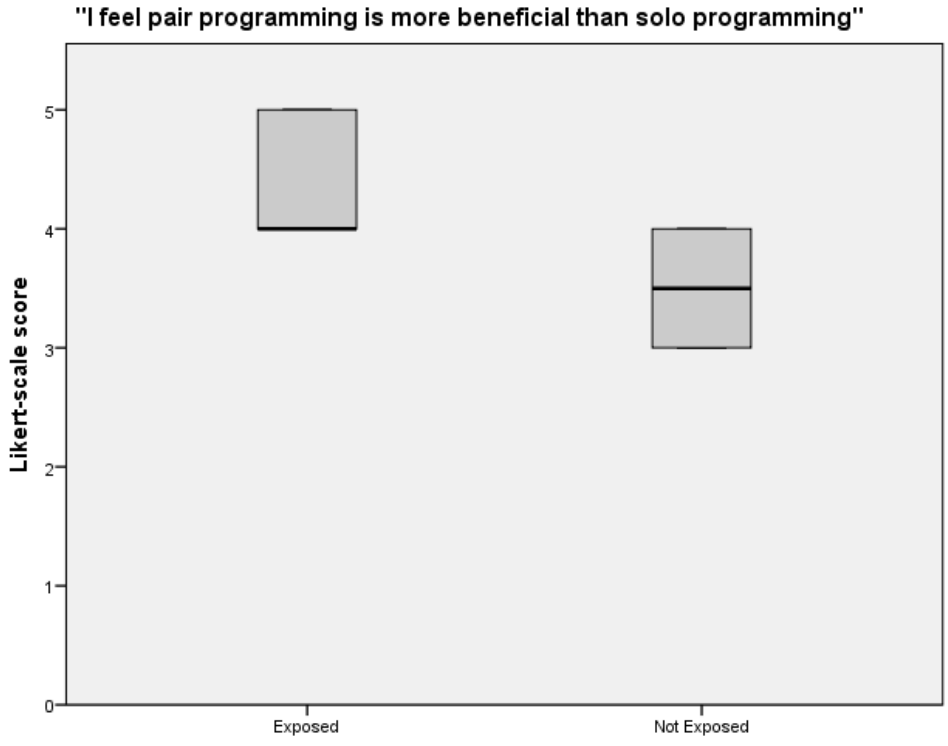


Figure 46: Reported scores for "I feel pair programming is more beneficial than solo programming".

The exposed group (M=4.3, SD=0.516) report higher scores than the control group (M=3.5, SD=0.58). There was no significant difference in perceived pair programming benefit between exposed students (*Mdn = 4*.0) and unexposed students (*Mdn* = 3.5), *U* = 4, *z* = -1.936, *p* = 0.114.

These results show that following the session, the overall student perception was that pair programming was more beneficial than solo programming.

6.4.2.4 Results

Ease of Communication

Table 28: Descriptive Statistics for Ease of Communication (Part 1B)

Ease of Communication	Exposed		Not Exposed	
	M	**SD**	**M**	**SD**
Part 1A (n=26)	4.6	0.51	3.9	0.90
Part 1B (n=10)	5.0	0.00	4.2	1.14

As this was the second time the students were working in pairs with the same partners, it was expected that the reported mean for ease of communication would be higher for both groups of students.

It can be seen that during Part 1B, the exposed pairs reported a higher mean for their ease of communication with a lower standard deviation (5.0 ± 0.00) when compared to Part 1A (4.6 ± 0.51). The students who were not exposed also reported a higher mean (4.2 ± 1.14 for Part 1B, compared with 3.9 ± 0.90 for Part 1A).

During Part 1B, the mean score for exposed students was higher with a lower standard deviation than the unexposed students, suggesting that despite the increased familiarity with each other, a second exposure to the pair programming guidelines was associated with improved intra-pair communication.

Figure 47 gives the distribution of scores reported by students for *ease of communication* between the two groups Part 1B of the study. It can be seen that the students who were exposed to the guidelines reported a higher score than students who were not, with no deviation.

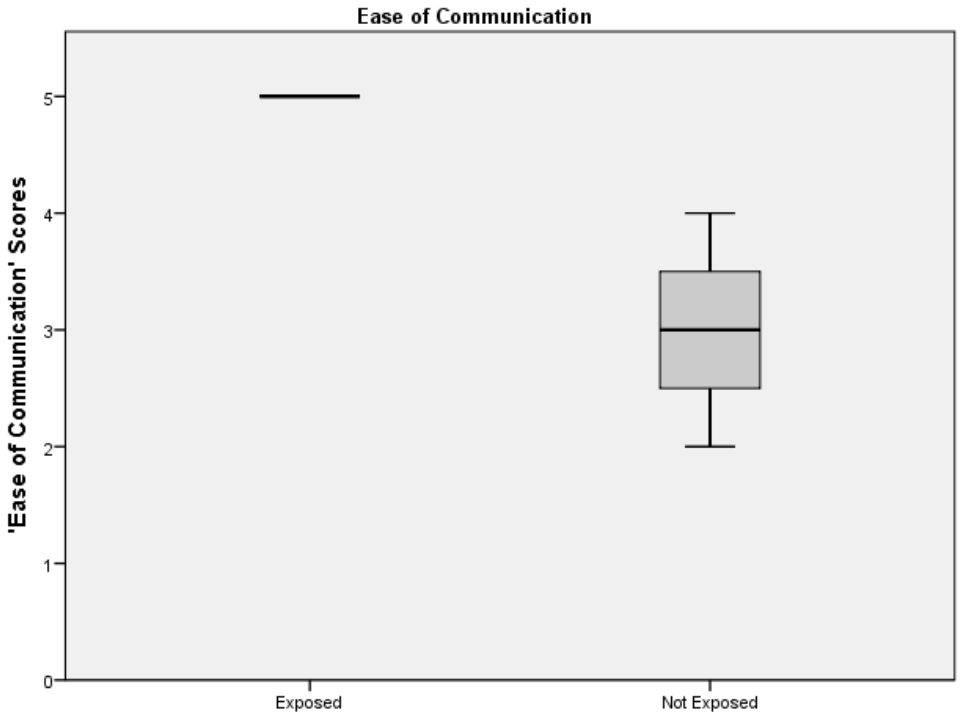


Figure 47: Reported scores for ease of communication

A Mann-Whitney U test was run to determine if there was a significant difference in Ease of Communication between the exposed and unexposed groups. There was a statistically significant difference in ease of communication scores between exposed students (*Mdn* = 5.0) and unexposed students (*Mdn* = 3.0), $U = 0$, $z = -2.893$, $p = 0.004$.

In this case, $p < 0.05$, confirming that there were differences between the two groups. The difference in median scores suggests that the exposed group found communication to be easier than the unexposed group.

Perceived Partner Contribution

Table 29: Descriptive Statistics for Perceived Partner Contribution (Part 1B)

Perceived Partner Contribution	Exposed		Not Exposed	
	M	SD	M	SD
Part 1A (n = 26)	4.8	0.43	4.2	0.84
Part 1B (n = 10)	4.8	0.41	4.1	1.20

During Part 1B, the exposed pairs reported a higher mean for their partner's contribution with a lower standard deviation (4.8 ± 0.41) compared to Part 1A (4.8 ± 0.43). The students who were not exposed, on the other hand, reported a lower mean, and a higher standard deviation (4.1 ± 1.20 for Part 1B, compared with 4.2 ± 0.84 for Part 1A), suggesting that some of the unexposed pairs did not feel that their partner contributed as much as they had during the first session.

Figure 48 gives the distribution of Likert scale scores for students' *perceived partner contribution* between the two groups for Part 1B of the study. It can be seen that students who were exposed to the guidelines rate their partner's contribution to be quite high, with low variance.

The asterisk indicates outliers in the data for three of the students.

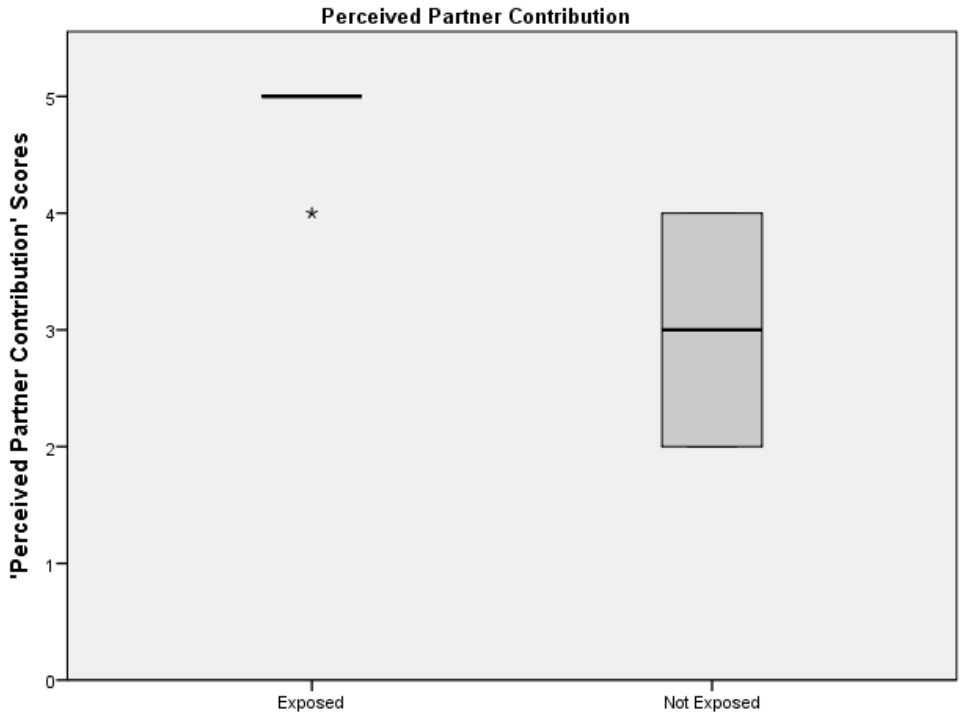


Figure 48: Reported scores for perception of partner contribution

A Mann-Whitney U test was run to determine if there was a significant difference in Perceived Partner Contribution between the exposed and unexposed groups. There was a statistically significant difference in perceived partner contribution scores between exposed students (*Mdn* = 5.0) and unexposed students (*Mdn* = 3.0), *U* = 1.0, *z* = -2.546, *p* = 0.011.

In this case, $p < 0.05$, therefore showing that there were significant differences between the two groups. The difference in median scores suggests that the exposed group found their partners to have contributed more than those in the unexposed group.

Successfully Completed Programs

Table 30 compares the mean and standard deviation of the pair's successfully completed programs between the two parts of the debugging study:

Table 30: Descriptive Statistics for Number of Completed Programs

Completion Rate	Exposed		Not Exposed	
	M	SD	M	SD
Part 1A (n=13)	2.7	3.04	2.2	2.14
Part 1B (n=5)	4.7	0.58	1.0	1.41

The Mean and SD in the table above show an improvement in performance by the exposed students in Part 1B, suggesting that a second exposure to the guidelines was somewhat beneficial to the pairs. In order to understand this, two statistical tests were applied to the data: (a) to compare the exposed and the unexposed group's results during Part 1B of the study, and (b) to compare the scores of the exposed groups between Parts 1A and 1B of the study.

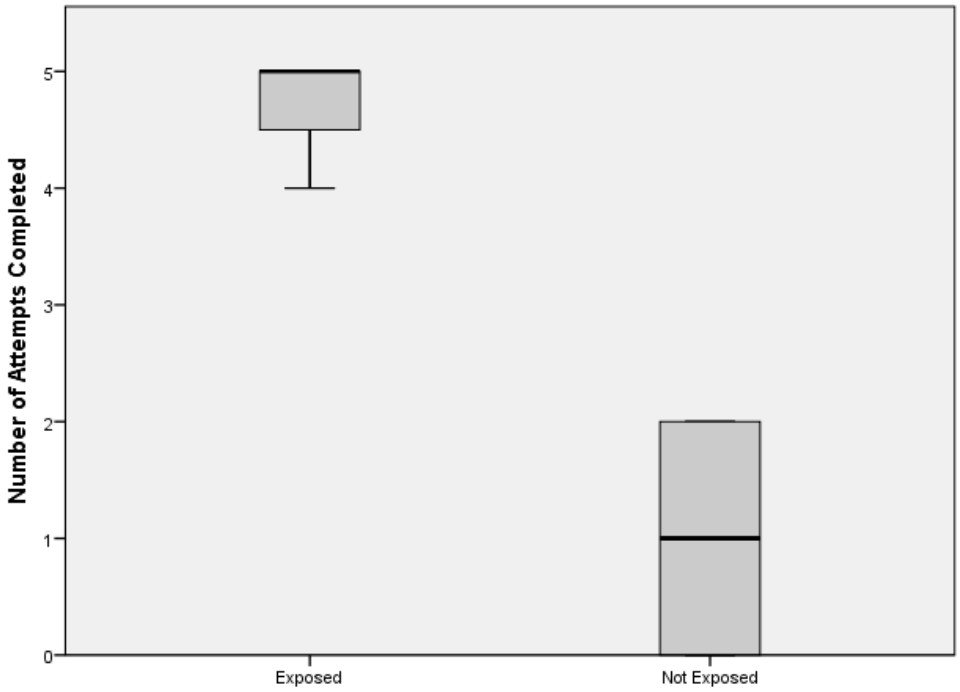


Figure 49: Number of tasks completed in Part 1B

First, an independent-samples t-test shows that there were no significant differences between the exposed students and the unexposed students during Part 1B of this study: t(1.227) = 3.479, p = 0.141. This indicates that exposed students did not perform significantly better than the unexposed group in this set of tasks.

The second test compared the scores of the five pairs who participated in Part 1B with the same pairs' scores from Part 1A to test for any differences using a paired-samples t-test. No significant differences were found when comparing Part 1B to Part 1A scores for exposed students: t(2) = -0.555, p = 0.635, or students who were not exposed: t(1) = -1.00, p = 0.500.

It is clear from visually assessing the data presented in Figure 49 above that the exposed pairs achieved a higher mean number of completed programs with a lower standard

deviation during Part 1B (4.67 ± 0.58), when compared to their overall results in Part 1A (2.71 ± 3.04).

The results reported for success in Part 1B are not statistically significant. This is in part due to the small sample size (n=5 pairs). When this is analysed, it is further split into two groups: exposed (n=3 pairs) and unexposed (n=2 pairs), leading to statistical data derived which cannot be said to be conclusive.

6.4.2.5 Limitations and Discussion

The preliminary results presented in Part 1A were subject to several limitations, some of which were addressed during Part 1B. In particular, Part 1B was carried out in an environment which allowed students to use compilers and debug the code, thus ensuring a process that was more natural to them. The analysed data allows for several conclusions to be made – however, these are subject to certain limitations and threats to validity.

This session consisted of five pairs (ten participants), leading to a very small sample size. Potential participants were limited to ones who had previously taken part in Part 1A, which allowed for comparisons to be made between the two study sessions. As a result, however the effects of exposure to the guidelines cannot be easily generalised.

The data suggests that the guidelines were somewhat beneficial in the context of debugging:

1. Students who were exposed to the guidelines reported significantly higher scores for ‘ease of communication’.

2. Students who were exposed to the guidelines reported significantly higher scores for ‘perceived partner contribution’.

However, the guidelines were created with the aim of helping pairs facilitate their communication during programming tasks. Furthermore, there is no statistical evidence to indicate that exposure to the guidelines had an impact on the pair’s success levels. Part 1B of this study asks students to debug existing code: the pair is therefore ‘pair *debugging’* and not ‘pair *programming’*, and the benefits identified here from the application of the guidelines cannot be said to apply to pair programming.

6.5 Part 2: Pair Programming Study

This section introduces a separate, final study to address the Part 1A and Part 1B limitations (a small sample size, and non-programming tasks) and reach conclusions about whether or not the guidelines can bring benefit novice student pairs in a programming context.

6.5.1 Study Design

One of the summer school programmes at the University of Dundee’s School of Computing uses a custom programming tool that has been developed to teach programming topics: the Abstract Programming Environment (APE). The APE tool runs on the NetBeans IDE, and provides a graphical front-end (Figure 50) which can be manipulated using Java code. This allows students to ‘see’ what they are programming. The contrast in Figure 50 has been adjusted to make the image more suitable for printing.

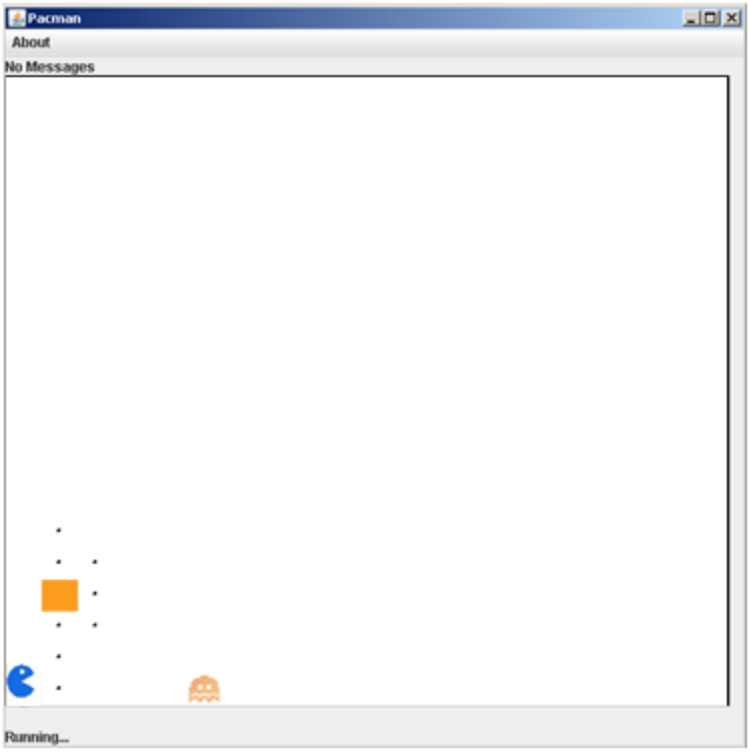


Figure 50: The APE graphical front-end

The APE tool consists of several challenges (or 'maps') in which students need to move the yellow character around, eating a number of dots; students must write this movement using Java code. Once all the dots have been eaten, the 'map' is considered complete, and students can move on to the next one.

The study was carried out during a four-week period, following the method described in section 6.2.

6.5.1.1 Materials and Equipment

Ten APE maps were chosen at random for the students to solve. As with Parts 1A and 1B of the study, all pairs were given a maximum time-limit of 45 minutes to solve as many maps as they could in a sequential order.

Pairs were provided with a list of basic instructions to move the character (Table 31), but were free to implement solutions using any programming technique at their disposal (e.g. in this study, students have used *for* loops and *do..while* loops to refactor the code. Some of the pairs were also observed to write a parser, which allowed for a more straightforward manner of telling the character how to move across the map).

Table 31: Basic instructions for the APE tool

Instruction	**What it does**
main.move();	Makes the yellow character move one space forward in whatever direction is being faced.
main.turnLeft();	Makes the character turn 90 degrees to the left.
main.turnRight();	Makes the character turn 90 degrees to the right.

Each pair is responsible for the whole programming process: from discussing possible solutions, to attempting to implement the correct code and testing it.

A five-question survey (Appendix I) was produced was based upon the survey used for Parts 1A and 1B of the study, with the additional question to find out about the role that each student had taken within the pair. It was used to collect data from the individual developers immediately after their debugging session. This data was used to determine if there was any significant difference between the groups that could bias the results.

6.5.2 **Participants**

Participants who had previously participated in Parts 1A and 1B were not deemed eligible for this study, as they had prior experience with the guidelines. A total of 28 participants were recruited (Level 1: 10 students; Level 3: 18 students), all of whom had previously used Java as a programming language throughout their courses.

Pairs were set up so that each pair consisted of students at the same level of study. Within each level, 50% of the pairs were randomly allocated to a group which would be exposed to the guidelines (n = 7 pairs), leaving the rest of the sample (n = 7 pairs) as a control group.

6.5.3 **Participant Experience**

Previous Programming Experience

As before, the students' reported experience with solo and pair programming were analysed to ensure no significant differences between the two groups which may have otherwise affected the data.

Table 32: Student programming experience

	Exposed		**Not Exposed**	
	M	**SD**	**M**	**SD**
Solo Programming Experience (years)	3.7	2.17	2.7	1.86
Pair Programming Experience (years)	0.3	0.59	0.2	0.41

As before, Mann-Whitney U tests were used to analyse the data.

The data show that the groups had somewhat different levels of experience; on average, more individuals in the "exposed" pairs had solo programming experience. Statistical tests were carried out to establish whether the differences between the two groups were significant and whether they might cause the results to be biased:

- No significant differences in ‘solo’ programming experience were found between the experimental and control groups: $U = 125$, $z = 1.266$, $p = 0.227$ ($p > 0.05$).
- Similarly, no significant differences in pair programming experience were found between the experimental and control groups: $U = 106.5$, $z = 0.427$, $p = 0.670$ ($p > 0.05$).

The results show that there were no significant differences between the two groups, and that further analysis should not be skewed by any bias resulting from one group having additional previous experience.

Perceived Benefits of Pair Programming

As part of the post-test survey, students were asked to rate the statement ‘*I feel pair programming is more beneficial than solo programming’* on a 5-point Likert scale.

Figure 51 charts student responses between the two groups:

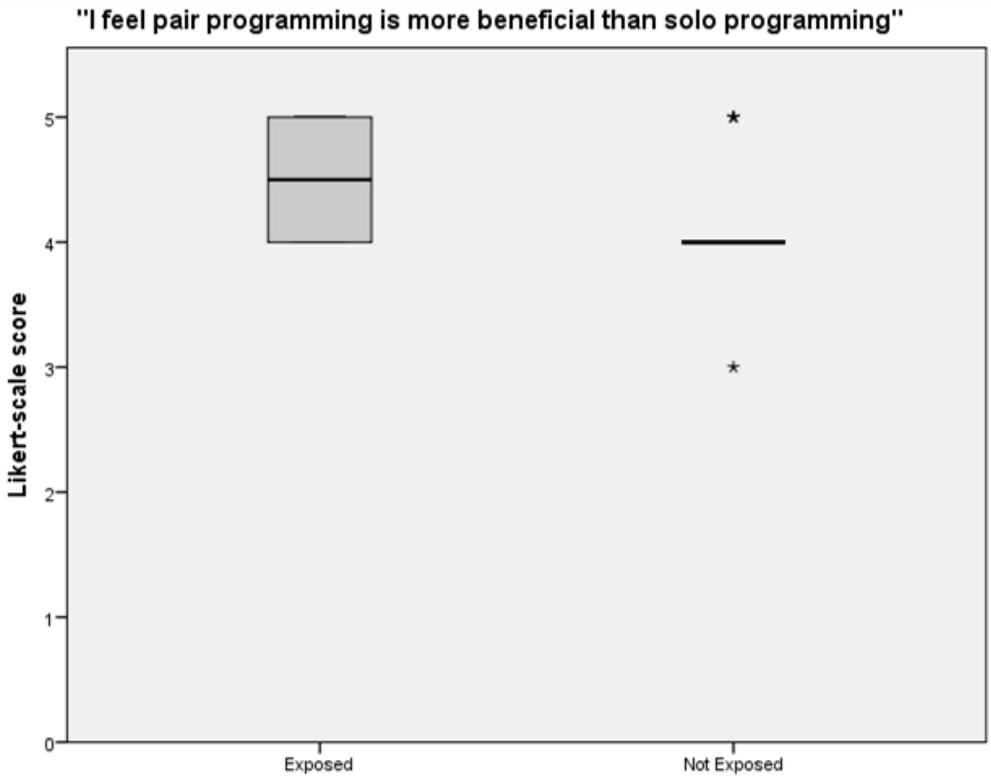


Figure 51: Reported scores for "I feel pair programming is more beneficial than solo programming".

The exposed group (M=4.5, SD=0.52) and the control group (M=4.1, SD=0.62) report similar scores. As observed in previous studies, there was no significant difference in perceived pair programming benefit between exposed students (*Mdn* = *4*.0) and unexposed students (*Mdn* = 4.5), $U = 133$, $z = 1.834$, $p = 0.067$.

These results show that following the session, the student perception was that pair programming was more beneficial than solo programming, regardless of whether they were exposed to the guidelines or not.

6.5.4 **Results**

The Likert scale data resulting from the post-test surveys were analysed to determine whether there were any significant statistical differences reported between the students who were exposed to the guidelines and those who were not.

As each individual completed their own post-test survey, the population consisted of 28 students, 14 of whom were exposed and 14 students who were not.

Shapiro-Wilk tests were carried out to understand whether the data being analysed were normally distributed. *Ease of Communication* scores for both exposed and unexposed groups were not normally distributed ($p < 0.05$). Similarly, scores for *Perceived Partner Contribution* for both groups were not normally distributed ($p < 0.05$). As the data are not normally distributed for both sets of scores, non-parametric tests were used.

Ease of Communication

The post-test survey results relating to *ease of communication* were analysed, and descriptive statistics were used to gain an overview of detail (Table 33).

Figure 52 depicts the distribution of scores reported by students for *ease of communication* (ranging from 1 (“strongly disagree”) to 5 (“strongly agree”)) between the two groups. The asterisk indicates outliers in the data.

Table 33: Descriptive Statistics for Ease of Communication (Part 2)

	Exposed		**Not Exposed**	
	M	**SD**	**M**	**SD**
Ease of Communication	4.9	0.27	4.0	0.78

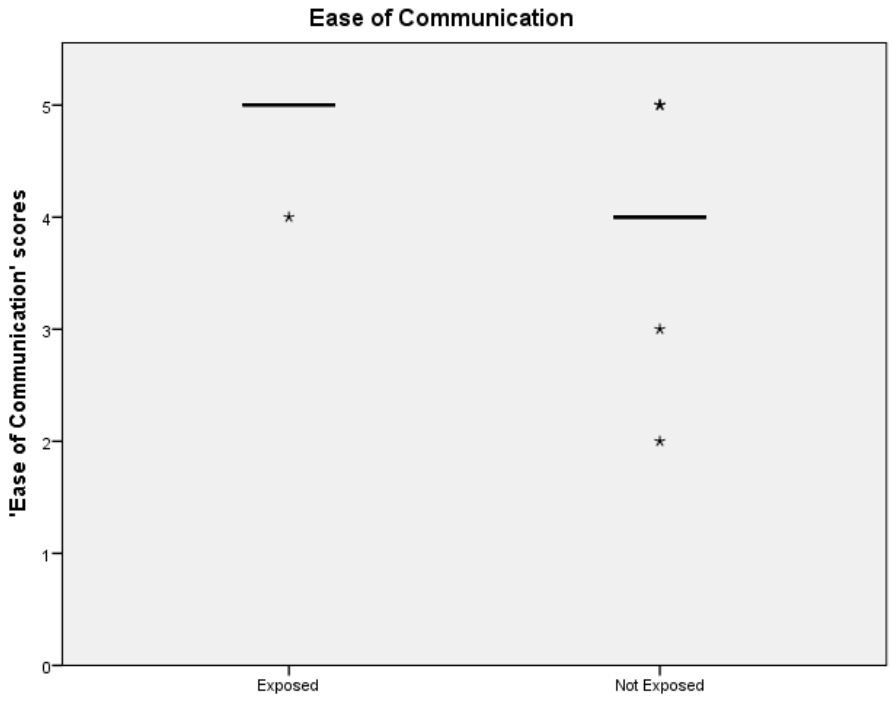


Figure 52: Reported scores for ease of communication

It can be seen that the students who were exposed to the guidelines reported a higher score than students who were not, with a lower variance.

A Mann-Whitney U test was run to determine if there were differences in Ease of Communication between the exposed and unexposed groups. There was a statistically significant difference in ease of communication scores between exposed students (*Mdn* = 5.0) and unexposed students (*Mdn* = 4.0), $U = 169$, $z = 3.721$, $p = 0.001$.

In this case, $p < 0.05$, therefore the null hypothesis (*the distribution of the pair's ease of communication is equal across the two groups*) was rejected.

Perceived Partner Contribution

As before, the post-test survey results relating to *perceived partner contribution* were analysed, and descriptive statistics were used to gain an overview of detail (Table 34).

Figure 53 shows the distribution of Likert scale scores for students' *perceived partner contribution* (ranging from 1 ("no participation") to 5 ("excellent")) between the two groups. The asterisk indicates outliers in the data.

Table 34: Descriptive Statistics for Perceived Partner Contribution (Part 2)

	Exposed		**Not Exposed**	
	M	**SD**	**M**	**SD**
Perceived Partner Contribution	4.9	0.36	3.9	1.07

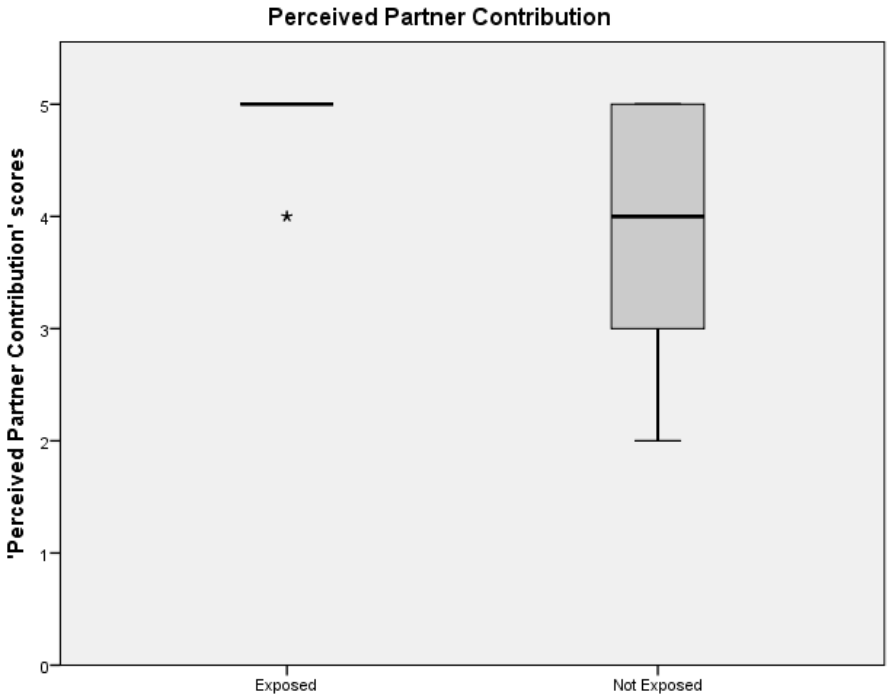


Figure 53: Reported scores for perceived partner contribution

It can be seen that generally, students who were exposed to the guidelines rate their partner's contribution to be quite high, with low variance.

A Mann-Whitney U test was run to determine if there were differences in Perceived Partner Contribution between the exposed and unexposed groups. There was a statistically significant difference in perceived partner contribution scores between exposed students (*Mdn* = 5.0) and unexposed students (*Mdn* = 4.0), $U = 146$, $z = 2.587$, $p = 0.027$.

In this case, $p < 0.05$, therefore the null hypothesis (*the distribution of the pair's perceived partner contribution is equal across the two groups*) was rejected.

Successfully Completed Programs

Following the test period, the number of tasks attempted was noted by the researcher, and scored at a later date. Each attempt was scored by the researcher, and also compiled, to see correct result was produced (i.e. if each map was solved successfully). The total number of successfully completed tasks was then noted for each pair.

An independent-samples t-test was run to determine if there were differences in completion scores between pairs who were exposed to the pair programming guidelines ($n = 7$), and those who were not ($n = 7$).

There were no outliers in the data, as assessed by inspection of a boxplot (Figure 54). The tasks completed for each level of exposure were normally distributed, as assessed by Shapiro-Wilks test ($p > 0.05$), and there was homogeneity of variances, as assessed by Levene's Test for Equality of Variances ($p = 0.903$).

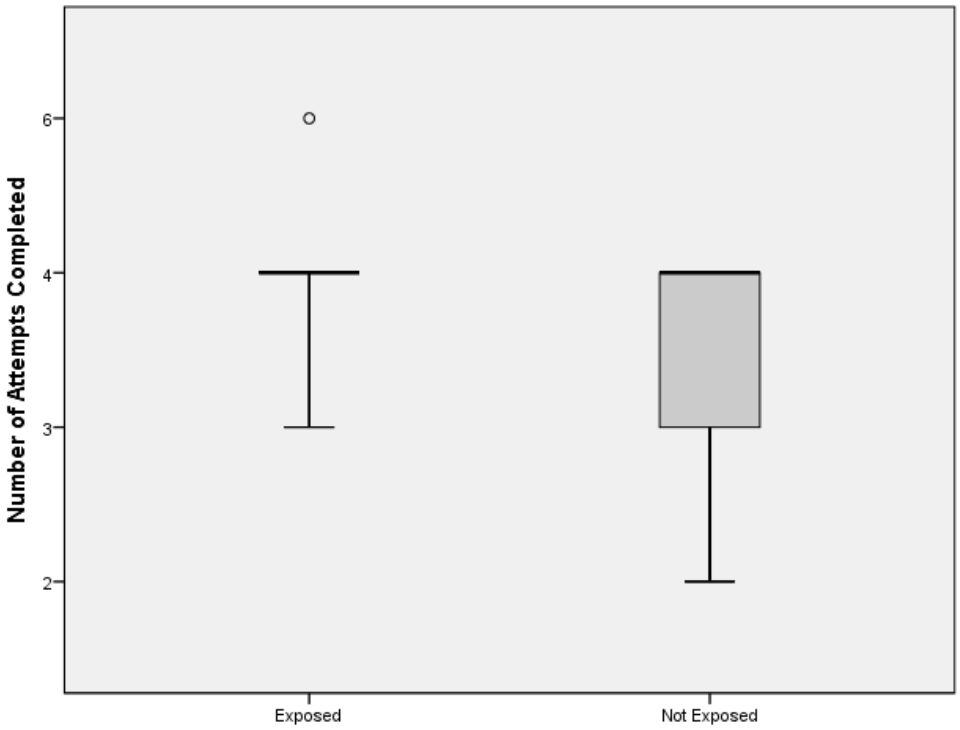


Figure 54: Number of tasks completed in Part 2

The exposed pairs completed a slightly greater number of tasks completed (4.0 ± 1.00) than the unexposed pairs (3.3 ± 0.76). The difference is not statistically significant; $t(12) = -1.508$, $p = 0.158$.

This result shows that exposing pairs to the guidelines does not increase their chances of successfully completing their tasks: exposure does not improve success rate.

6.5.5 **Indicated Preference of Driver-Navigator Role**

In Chapter 5, it was reported that "the ratio of students who preferred to drive against students who preferred to navigate was 55:45". As part of the post-test surveys for this study, students were asked to indicate which role they had experienced for the duration of the session of the study. Results were as follows:

- 9 students indicated that they were drivers;
- 11 students indicated they were navigators;
- 8 students ticked both boxes, indicating that they experienced both roles during the session.

This data shows that more students indicated a preference for the navigator role over the driver role, with an approximate ratio of 45:55.

The data was then explored on a ‘pair-by-pair’ basis, giving the following results:

- 9 pairs consisted of a driver and a navigator;
- 2 pairs consisted of a navigator and an individual who indicated they had experienced both roles;
- 3 pairs consisted of both members within the pair indicating they experienced both roles.

The first and last responses are consistent with the more traditional pair programming setup, and with what students are taught: a pair consists of a driver and a navigator, and these roles should be switched often.

The second statement does not fit this pattern, showing that whilst one member of the pair was a permanent navigator, the second member of the pair found it necessary to switch between the two roles. A review of the audio files was performed. It revealed that in both cases, the driver would sometimes stop typing, and brainstorm possible solutions and next steps with the navigator. Following this, they would go back to driving the session. It is possible that during these brainstorming sessions, the driver felt

that were also navigating, and thus felt they had experienced both roles during the session. It is unclear as to why the driver felt the need to switch back-and-forth between the roles, or why their navigator did not take over the driver role, but this hints at possible pair programming dynamics that may exist outside of the traditional 'driver-navigator' claim.

6.5.6 **Discussion**

The data gathered from this study supports the following hypotheses:

1. The distribution of the pair's ease of communication scores differs by exposure to the guidelines; i.e. pairs who were exposed to the guidelines reported significantly higher scores for *ease of communication* than the control group.

2. The distribution of the pair's perceived partner contribution scores differs by exposure to the guidelines; i.e. pairs who were exposed to the guidelines reported significantly higher scores for *perceived partner contribution* than the control group.

3. The mean number of completed tasks for pairs who were exposed to the guidelines and pairs who were not exposed is equal in the population; i.e. there was no significant difference in the number of completed programs between pairs who were exposed to the guidelines, and the control group.

These results and the accepted hypotheses are preliminary, but they show that the guidelines may help improve students' experienced communication within their pair. It is posited that this stronger 'partner contribution' was due to the fact that individual members of the pair are more confident communicating their ideas (possibly due to the additional advice provided by the guidelines); in turn, to their partner, it seems as if they

are making more contributions during the pairing session. Furthermore, the use of the guidelines may support students in dealing with issues and barriers that typically arise during pair programming sessions in a structured way. However, whilst these guidelines can be seen to aid the pairs' perceived communication, there is no evidence to suggest that the guidelines have any impact on student success.

These findings are limited by the subject sample (from a single institution), and a relatively small sample group. A sample size of 28 participants gives a margin of error of 18.51% (CI: 95%). The margin of error could be reduced by running this study with more participants (e.g. with 50 participants, the margin of error drops to 13.84%). Increasing the sample size could give evidence to further support these conclusions, and allow these results to be further generalised beyond the scope of this study. This is further discussed in the Future Work section of Chapter 8.

6.6 Summary

The first two studies described in this chapter do not consist of pair 'programming'; instead, the pair is tasked with debugging the code through various methods (code reviewing, and using the compiler). This means that results that emerged from Parts 1A and 1B are not necessarily indicative of the effect the guidelines have on the pair's experience of programming. Furthermore, Part 1A of the study disallowed the use of compilers to promote discussions within the pair. However, there is insufficient literature to support the use of pen-and-paper systems in this manner, and there are no discernible advantages of doing this over more traditional debugging methods (as seen in Part 1B).

Part 2 of the study aimed to address these limitations: the task given to participants was a programming one, and more participants were recruited.

When analysing the Likert scale data, the results from Part 2 show that:

1. Students who were exposed to the pair programming guidelines reported that their communication was easier than the students who were not exposed, and;

2. Students who were exposed to the guidelines reported higher perceived contributions from their pair partner, when compared to the responses of students who were not exposed.

Further analysis also show that:

3. The guidelines do not have any significant impact on student success; moreover, the added communication did not seem to affect the experienced success levels. The guidelines are therefore not seen to significantly affect the groups' performance in successfully completed tasks.

The findings in this chapter suggest that the pair programming guidelines can be used to improve the way novices communicate within their pairs. These results indicate that the guidelines are useful: it was reported that the pair's communication seemed to be easier, and that the students' partner's contributions to the session seemed to be stronger.

These results might not readily be generalised, and perhaps can only be used to suggest sources of improvement, rather to establish them. Nevertheless, if the results are to be generalised, Part 2 must be replicated with more subjects in diverse settings – this is discussed in more detail in the future work section of Chapter 8.

Chapter 7: Review of the Guidelines

In previous chapters, it can be seen that students who were exposed to the pair programming guidelines reported that they had experienced a higher ease of communication and higher levels of partner contribution than their non-exposed peers. This chapter reviews feedback and opinion about the guidelines gathered from two user-groups: students who were exposed to the guidelines in previous studies and industry members.

7.1 Gathering Feedback

Two groups were surveyed: (i) the students who had been exposed to them, and (ii) pairs in the industry. The latter can be said to consist of two further subgroups; industry-based pairs who had been previously observed, and industry-based members with no previous association to the project. The groups were asked to comment on their use and thoughts regarding the guidelines, with the aim of exploring which aspects were valuable, and why.

7.1.1 Comments from Students

All students involved in the studies described in Chapter 6 who were exposed to the guidelines were invited to give detailed feedback on their experiences by means of an online survey, built using the Bristol Online Surveys tool[9].

The survey consisted of questions that ask students whether they had used each of the pair programming guidelines during the study. For each question, if the student ticked the ‘yes’ box, they were asked to give more detailed feedback on their experience with

[9] http://www.survey.bris.ac.uk

that particular guideline. Figure 55 shows an example screen shot from this survey; Question 1 does not show the opinion box, as the participant ticked 'no'.

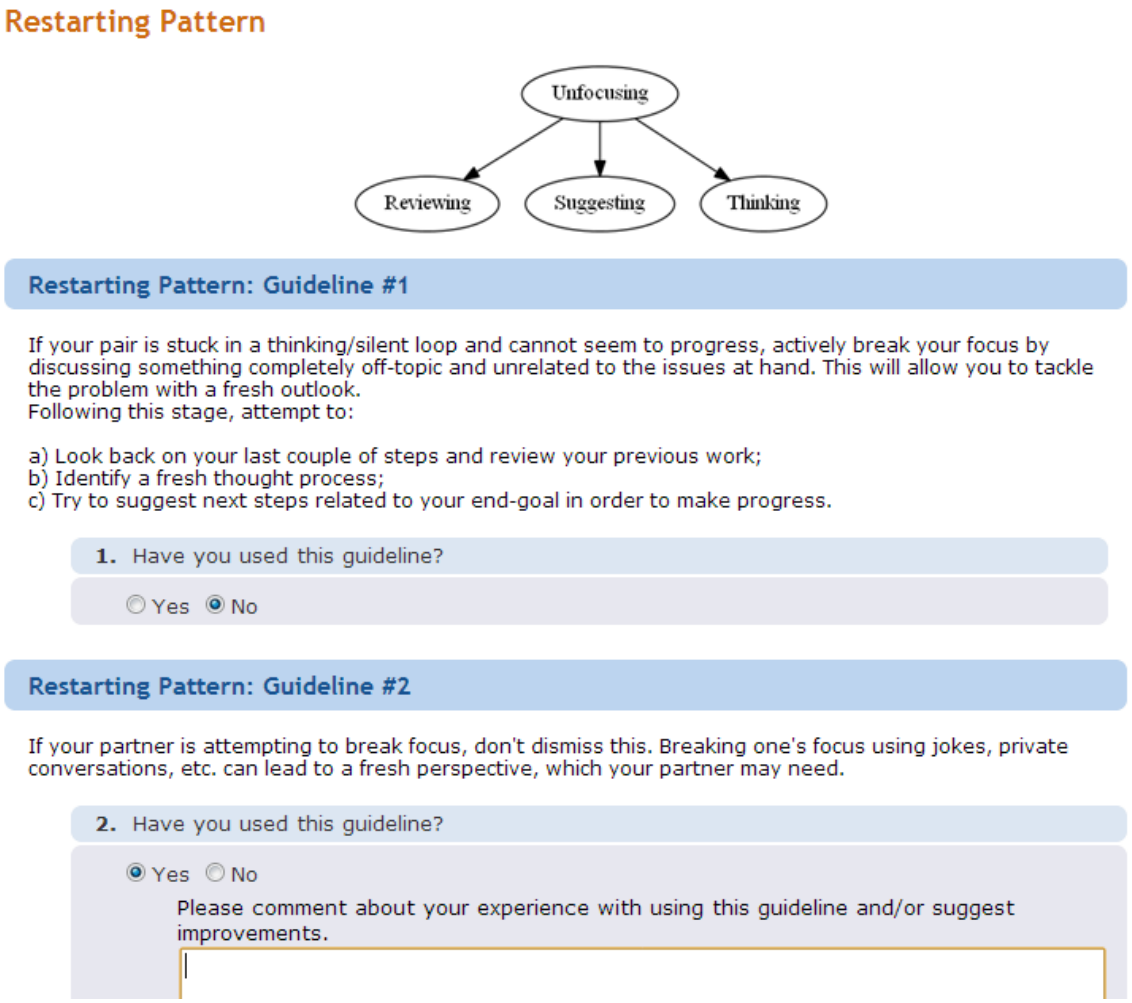


Figure 55: A screenshot from the online survey

A total of six students completed the survey. Their reported usage of the guidelines is given in Figure 56. This is a rather small sample size, further limited by the fact that arguably, students who replied to the survey may have been the ones who were most interested in pair programming (thus creating a potential bias in the results). The discussion that follows is not representative of the general population, but can be used to understand how each guideline was perceived and used by the respondents.

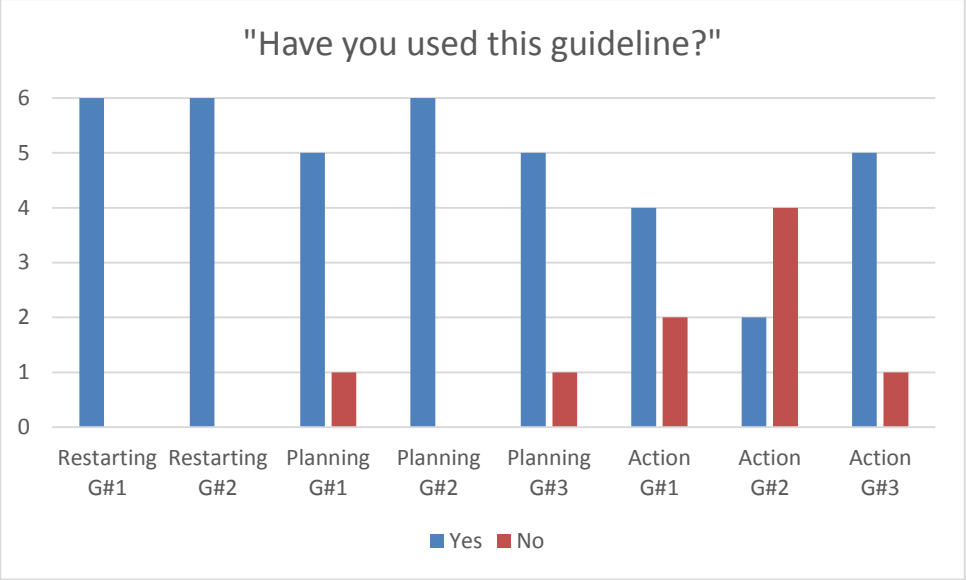


Figure 56: Usage of guidelines

It can be seen that seven of the eight guidelines were used by most or all students who completed the survey. The next section will discuss feedback regarding the individual guidelines.

7.1.1.1 Feedback regarding Restarting

- RESTARTING G#1: *If your pair is stuck in a thinking/silent loop and cannot seem to progress, actively break your focus by discussing something completely off-topic and unrelated to the issues at hand.*

All students who answered the survey indicated that they had made use of this guideline whilst pair programming; "*When we were stuck, we lost focus and ended up going off-topic anyway before bringing it back to the task at hand*" / "*It was helpful to go for a walk, and then return less frustrated*."

Comments about the guideline were positive – "*This was a useful technique*" / "*Quite happy with using it; worked well*", with some students suggesting that this guideline tended to occur naturally to them, without much planning required – "*Tended to use this naturally*" / "*We both used it intuitively without thinking about it*".

- RESTARTING G#2: *If your partner is attempting to break focus, don't dismiss this. Breaking one's focus using jokes, private conversations, etc. can lead to a fresh perspective, which your partner may need.*

All students indicated that they used this guideline whilst pair programming, indicating that they found this to "*usually work quite well*", commenting that the "*use of jokes or venting frustrations were helpful towards giving us a break*".

Some comments focused on the possibilities of having problems when working with a new partner: "*I can imagine if you do not know your partner very well it would be more tempting to dismiss an attempt to break focus*" / "*If one of us lost focus, the other was generally losing focus at the same time*". Other comments, on the other hand, discuss possible solutions to this issue: "*Sometimes identifying when your partner wishes to break focus for this purpose can be difficult, particularly if you do not know your partner well or feel uncomfortable working with them. Having said this, it can be as simple as merely saying, 'let's take a quick break' – this can provide a clear indication of intentions."*

7.1.1.2 Feedback regarding Planning

- PLANNING G#1: *Suggestions and reviews are optimal states that will allow you to drive your work forward. When in these states, feel free to alternate (e.g.*

review previous code, suggest an improvement, review methods to be changed, suggest potential impact).

Out of the six students who answered the survey, five had used this guideline whilst pair programming. Feedback was positive – students felt that this guideline "*had a natural flow*", and that "*group coding would be impossible without this*".

The student comments reflected on their experience using the guideline, showing an understanding of the concept behind it: "*when we were completely stuck, we would look at previous code and try to work through in our heads what it was supposed to be doing*"; and "*constantly reviewing the work in progress helps to provoke new thoughts or improvements*".

- PLANNING G#2: *At each stage, do not hesitate to ask your partner for clarification as to what they are working on, or suggesting.*

All students indicated that they used this guideline.

Comments were positive; students reported that "*[it was] helpful to know what [their] partner was thinking*", and explained that "*this can help to ensure that [both partners] understand the on-going work and are on the same page*."

Once again, student comments showed an underlying understanding of the concepts introduced by this guideline: "*this helps the person with the idea 'concrete' it in their head, and lets the partner get a new angle on the problem to improve the solution, or find flaws*." / "*This can be used to spot errors in logic while the partner is explaining it out loud*."

- PLANNING G#3: *Think about what your partner is saying and doing. Offering an explanation of the current state can help move the work forward.*

Five students out of six indicated that they had used this guideline. Comments were positive and similar to previous comments for the planning guidelines; students felt that this guideline enhanced their teamwork and ensured an in-depth understanding of the code.

One student was less positive, and felt that "*asking what the partner is doing at every stage can be irritating and detrimental; sometimes it's best to sit back and watch*", suggesting that for some students, the constant offer or request for explanations might prove to be distracting. This shows that in some cases it might be better for the pair to discuss the guidelines between themselves prior to adopting them, and develop a way for them both to be comfortable with their usage in terms of distractions and interruptions. It also shows that the guidelines cannot be applied unthinkingly, but must be used with some sensitivity to the overall context.

7.1.1.3 Feedback regarding Action

The Action set of guidelines was the least used by the survey respondents. This could be due to the fact that this set of guidelines has limited applicability, as each guideline is particularly targeted towards an individual member of the pair (either the driver or the navigator). Therefore, if the survey respondents had not encountered the described situation whilst in a certain role, they would have had no opportunity to experience these guidelines.

- ACTION G#1: *Navigator: Whilst the driver is coding, actively look to make suggestions that contribute to the code.*

Four students out of a possible six indicated that they had used this guideline, leaving positive feedback about their experience. Students felt that reading the code as it was being typed by the driver "*helped save time*", and agreed that following the code allowed them to be more proactive when helping, as they could make suggestions when the driver appeared to be struggling.

- ACTION G#2: *Navigator: If the driver is muttering, use this opportunity to make sure your suggestions have been properly understood.*

This guideline was the least used, with only two of the survey respondents indicating that they had used it. Both students commented positively that that the guideline "*works well*" and that it helped them with getting extra clarification when they were stuck while the driver was muttering.

- ACTION G#3: *Driver: Whilst you are programming, or thinking about your code, voice your thoughts (even if it's just mumbling and muttering while you're typing).*

Five out of six students indicated that they had used this guideline. The reported feedback was positive: students expressed that their muttering "*helped keep the navigator involved and encouraged them to contribute*", meaning that at times, the problem is solved "*before you waste time getting neck-deep in useless code*". Comments suggest that this guideline was considered beneficial in helping the survey respondents understand the underlying logic behind their code.

7.1.2 **Comments from Industry Members**

The guidelines were also distributed to industry-based developers (with more than 6 months' experience of pair programming) to obtain feedback based on their more extensive experience.

A distribution list consisting of the observed pairs from Chapters 3 and 4, contacts made by the researcher from various networking events and conferences, and agile groups (such as the AgileAlliance[10] and the BCS Agile Methods Specialist Group[11]) was e-mailed an A4 sheet consisting of the patterns and guidelines, specifically asking for feedback on the guidelines as follows:

> "As a result of my research, I have produced a set of guidelines which aim to help novices communicate better within their pairs. Your feedback would be appreciated."

A total of 24 industry members replied, 18 of whom had participated in the observations reported in Chapter 4 and whose feedback is available in Appendix J. For the purposes of this chapter, comments from previously observed developers will be noted as "Group A", and comments from independent developers will be noted as "Group B". Comments were returned via e-mails consisting of either text pertaining to each separate set of guidelines, or a scanned version of the A4 sheet with annotations made by the developers.

The following sections will review the industry feedback received.

[10] http://www.agilealliance.org/
[11] http://www.bcs.org/category/16392

General Feedback

General comments suggested that industry members confirmed they practiced driver-navigator role-switching, which is often seen as part of the general practice: "*switching regularly between roles keeps both members sharp and involved*." Furthermore, developers were aware that within their own work, they had experienced behaviour similar to that described by the guidelines: "*I've definitely seen and been in all three of those situations before*."

Feedback on Restarting

All comments were given on the *Restarting* concept, rather than on each specific *Restarting* guideline. Many industry members agreed with this set of guidelines: "*This is good advice*" or "*I agree with breaking focus – it can be tiring*".

Group A thought that this was "*good advice*". Developers likened this set of guidelines to concepts they were familiar with, such as the Pomodoro timer[12], a time management technique where the person is asked to force a break (or an 'unfocusing') episode at set intervals of time. Their feedback also focused on the 'unfocusing' stage in agreement, with suggestions on how to successfully execute it. The majority of comments suggested physically walking away from the desk and "*going for a coffee*" and "*snacking/drinking at these times*" as a useful way of 'unfocusing'. One person suggested "*bringing in a third person*" during these breaks, to help gain perspective on the current problem. It was also suggested that these breaks should be taken "*just after writing a failing test [..] so when you get back to work, you know where to continue*".

[12] http://www.pomodorotechnique.com/

Developers from Group B discussed strategies to work through potential implementation issues: "*suggest next steps to help avoid over-engineering*" / "*try to decompose a particular [problem] into smaller steps*". The notion and importance of a pair member knowing *when* to speak (as seen in the comments from the observed developers) was reiterated: "*Give [your partner] some space to read the code himself before making suggestions*."

Comments also cautioned against breaking focus too often as *"it's easy to lose focus*". This was also reiterated by industry members outside of the observed group; "*When you're trying to think something through, there is 'social pressure' to continue to talk, when thinking quietly could be more useful – we almost never 'go silent'."* Comments such as this underline the importance of understanding that silence – and having enough time to think as an individual – is an important aspect of communicating with your partner.

Finally, two comments reiterated the guidelines. The first of these comments in particular is a succinct summary of the *Restarting* guidelines. The second comment, on the other hand, is a recommendation for the *Action* set:

> "*Don't dismiss your partner trying to break focus.*"
>
> "*Giving voice to your thoughts might help*."

***Feedback on* Planning**

All comments were given on the *Planning* concept rather than on specific guidelines.

Typically, comments were in agreement with this concept: "*I totally agree with benefits of discussion, clarifying motivation, etc.*" It was typically felt that the guidelines within

the *Planning* concept would help to minimise disruptions and stop pairs from going off at a tangent to the task at hand; "*This does help many times – mostly to realise that you're on the wrong track*."

Group A recognised the guidelines as describing aspects of their own work-patterns: "*We typically* do *review and explain – especially when this is the first activity of a pair with a new member*." These developers drew parallels between this set of guidelines and Test-Driven Development (TDD). TDD is a software development technique that relies on short cycles: the developer starts out by writing a failing test case which outlines a new addition to the code, and then proceeds to write code which will allow the test to pass (Beck, 2003). Comments from developers show that this set of guidelines "*fits in with TDD (write test, pass test, refactor), with each phase providing an opportunity to switch the driver*" / "*alternating the keyboard after each test implementation combo keeps both partners in sync*."

Developers also commented based on their previous experiences, stressing that the continuous feedback emphasised by these guidelines was important: "*learning to say 'I don't know' or 'I don't understand' is critical*". Group B were in agreement with this. One comment expands on the guideline, further emphasising the need for immediate explanation:

> "*Sometimes while I'm coding, and the Navigator asks what I'm doing, I find myself saying, 'you'll see in a while'. I think this should be avoided. Try to explain NOW. This can be good for both developers."*

Within this set of guidelines, industry members also felt that it was necessary to note down various ideas and action plans; "*capturing as many suggestions as possible is*

particularly helpful if we need to backtrack". A perceived advantage of this is that discussions are not unnecessarily repeated over and over when coding similar tasks – a log should ideally act as a buffer between proposed ideas and working solutions.

***Feedback on* Action**

All comments were based on the *Action* concept rather than on each specific guideline.

Typically, comments by industry-based developers indicate agreement with the guidelines presented here: "*It is really useful – [discusses] the best part of PP.*"

Group A were divided regarding the muttering stage. Some developers felt that "*the driver should articulate what he is doing and thinking, not mutter*"; furthermore, "*sometimes you need to type – and explain afterwards*". On the other hand, other comments indicated that muttering "*helps your partner not get distracted*", and perhaps more importantly, "*stops the navigator from interrupting a train of thought*".

There were some mixed feelings regarding the 'voice your thoughts' part of the guidelines: some developers stated that "*this stops the navigator from interrupting a train of thought*" and that it "*helps your partner not get bored/distracted*", whereas other developers were in disagreement: "*sometimes, you need to type first, and explain afterwards*".

Group B suggested that due to the regular switching between roles, both the driver and the navigator needed to focus on their upcoming roles: the navigator should "*try to think ahead since [they'll] be the driver soon*", whereas the driver should "*fix problems spotted by the navigator ASAP to help them think about other issues*".

Another comment indicates that suggestions should be architectural, rather than focusing on errors which can be seen in the development environment's error console (e.g. 'you missed a semi-colon'); suggestions should be about consequences of the current approach.

Finally, the following developer (from Group A) summarises the concept behind this set of guidelines:

> "*This is good. I have had some silent partners and it tends to cause frustration as unless you know the pair very well, silent partners just look like they're clicking randomly on the screen.*"

7.2 Updating the Guidelines: version 1.5

The comments provided by skilled developers suggest several additional guidelines which were grouped together. The key points were summarised into additional communication guidelines, presented in sections 7.2.1, 7.2.2 and 7.2.3 below.

7.2.1 Additional *Restarting* Guidelines

- If you are in disagreement with your partner, you may find it helpful to break for lunch/coffee/etc. – during which you should physically walk away from your desk.

- Give your partner space to read the code before suggesting future steps.

7.2.2 Additional *Planning* Guidelines

- Learning to say *I don't know* or *I don't understand* is critical.

- Always explain things immediately – try to avoid replying to a question with "you'll see in a while", as this will distract your partner.

- Make a note of previously discussed suggestions and reviews so that similar discussions are not unnecessarily repeated over and over.

7.2.3 Additional *Action* Guidelines

- When silent, it can look as if you are clicking randomly on the screen, which risks your navigator becoming bored and distracted. Voicing your thoughts can help counter this.

- NAVIGATOR: Think ahead, since you'll be driving in a short while: what is the current course of action not covering? Is there anything worth verifying that might have been left out?

These additional guidelines can be seen as an addendum to the original guidelines, with the aim of helping novice developers communicate better within their pairs.

7.3 Limitations

Feedback gathered from both groups is positive; however, this work is limited by several factors, which are discussed in this section.

The student group consisted of a very small sample, which cannot be considered to be representative of the general population (6 out of 34 exposed students filled in the survey). This is further limited by the fact that the students who did reply may be the ones who were most interested in pair programming and/or the provided guidelines, thus leading to comments that may be positively skewed. To counteract this limitation, feedback gathered from the student group was not used to inform further guidelines. Instead, this feedback was used to develop an understanding of how the guidelines are perceived and implemented by this group.

The response rate for the industry-based group showed that 18 of the 24 responses were obtained from pairs who had been involved in the creation of the guidelines through previous observations and discussions. These comments were condensed into additional guidelines. These pairs were less 'independent' of the guidelines, and they may have felt a certain degree of familiarity or ownership towards them, leading to comments that may have been positively skewed. Nonetheless, these comments still need to be evaluated; this is discussed in the further work section of Chapter 8.

7.4 Summary

An understanding of how the guidelines were used and perceived was developed by gathering and examining feedback gathered from students and industry members.

Students who had used the guidelines indicate that they used some more often than others, suggesting that some guidelines might have limited applicability depending on the context in which they are used. Furthermore, comments made by students suggest that the guidelines cannot be applied unthinkingly: it would be beneficial for pairs to discuss the guidelines between themselves prior to adopting them, in terms of potential interruptions.

Feedback gathered from industry members showed an endorsement of the guidelines presented, suggesting that developers had previously experienced situations described by the guidelines. This user group provided insight and comments on the existing guidelines, which were used to extract further guidelines. These are provided as an addendum to the existing set.

Chapter 8: Conclusions and Further Work

The final chapter summarises the contributions of this thesis and reviews findings from the observations carried out with industry experts and from the student evaluations. These are related back to the research question raised in Chapter 2. Finally, implications of these findings and suggestions for future work are considered.

8.1 Thesis Summary

An approach informed by grounded theory was adopted to explore and analyse how industry-based expert pairs communicated verbally. Observed pairs were seen to experience the following communication states:

- Reviewing
- Explaining
- Muttering
- Silence
- Code Discussion
- Suggestion
- Unfocusing

These states and the way observed pairs transitioned between them were analysed (Chapter 4), and resulted in the extraction of communication patterns. These patterns were then used to inform the creation of guidelines, which were designed to help novice pairs improve their experience of communication.

The remainder of the thesis describes a set of studies designed to investigate the efficacy of the guidelines. A qualitative analysis of initial reactions to the guidelines from novice student pairs showed positive feedback, indicating that these guidelines were seen as beneficial (Chapter 5). The quantitative evaluations reported in Chapter 6 present further results, indicating that novices who were exposed to these guidelines reported a better communication experience within their pair than those who were not exposed. Whilst the thesis is concerned primarily with novice communication, the effectiveness of the guidelines on student success (measured by analysing the numbers of problems solved in each particular task) was also measured (Appendix J), showing there was no significant impact on student success. In summary, the guidelines may be a helpful tool for novice pairs, with potential benefits including making their experience of communication seem easier and their partner's contributions seem stronger.

8.1.1 Research Question Revisited

The research question of this thesis was: *can extracted communication patterns from expert pair programmers be used to help novice student pairs to improve their intra-pair communication?*

As discussed in Chapter 2, developers expect certain attributes from potential pairing partners. Most notably, a good partner is expected to communicate well (Begel and Nagappan, 2008). However, pairs are sceptical about the added communication aspects required during their first pair programming experience (Williams et al., 2000), and communication is frequently seen as one of the top issues faced by novice pairs (VanDeGrift, 2004, Sanders, 2002).

The research question was addressed in two parts:

- Part 1: Extracting communication patterns from expert pairs.

This part of the research question was addressed in Chapters 3 and 4. In the context of this thesis, ‘expert pairs’ were defined as industry-based pairs with a minimum of six months of full-time commercial pair programming experience. An informative study was carried out using a set of videos recorded by one expert pair, the results of which were used to create a coding scheme. This coding scheme was applied to several other pairs within industry, and the frequency of analytic codes was investigated to understand the transitions between communication states. This understanding was depicted as communication patterns, and re-worded as guidelines for novice pairs.

In answer to the first part of the research question, it was possible to extract communication patterns from expert pairs.

- Part 2: Understanding if the patterns could help novice student pairs to improve their intra-pair communication.

This question was addressed in Chapters 5 and 6, with additional feedback forming part of Chapter 7. Through a set of qualitative and quantitative studies conducted with novice pairs, it was seen that novice student pairs who were exposed to the guidelines reported better communication than pairs who were not exposed to the guidelines. The individual’s experienced communication was measured using self-reported scales which queried novices on their experience with (i) ease of communication, and (ii) perceived partner contribution, based on the earlier literature review (in Chapter 2) which summarised these as the key problems for novices starting to pair program. The results obtained suggest that the industry-inspired guidelines may be useful for making this communication seem easier and also in making partner contributions seem stronger.

8.2 Thesis Contributions

- A coding scheme has been identified that can be applied to analyse pair programming communication. The codes were derived from observing and examining expert pair communication;

- Patterns of communication were identified based on the application of the coding scheme to observation sessions with industry-based pairs. These patterns were cast into pair programming guidelines;

- The guidelines have been evaluated with student pairs. This showed that exposure to these guidelines improved the self-perceived communication experience of novice pair programmers, but had no significant impact on their success levels.

8.3 Thesis Output

The set of guidelines developed throughout this thesis (including the addendums made in section 7.2 above) are summarised in Table 35. A website was created as an online repository to make the guidelines publically available: www.pairprogramming.co.uk. This was created with the aim of introducing the guidelines to people who may be interested in either introducing pair programming to their workplace, or implementing them as an accompaniment to their usual pair programming techniques. By making the guidelines publically accessible, it is intended that they will continue to be adapted and revised through continual feedback and interaction with the wider community. Following the launch of the website, representatives from the following educational institutes have expressed an interest in using the guidelines as part of their teaching programmes: Boston University, Northern Kentucky University, University of Aberdeen, KTH Royal Institute of Technology, Stockholm.

Table 35: The pair programming guidelines (version 1.5)

				Additional Guidelines	
Restarting **Guidelines**	If you and your partner are stuck in a silent period and cannot seem to progress, actively break your focus by discussing something completely off-topic and unrelated to the issues at hand. This will allow you to tackle the problem with a fresh outlook.	Following this stage, attempt to: - Look back on your last couple of steps and review your previous work; - Identify a fresh start; - Try to think about your end goal when suggesting next steps in order to make progress.	If your partner is attempting to break focus, do not dismiss this. Breaking one's focus using jokes and private conversations can lead to a fresh perspective, which you and your partner may need.	If you are in disagreement with your partner, you may find it helpful to break for lunch/coffee/etc. – during which you should physically walk away from your desk.	Give your partner space to read the code before suggesting next steps.
Planning **Guidelines**	Suggestions and reviews are both useful states that will allow you to drive your work forward. When in these states, feel free to communicate about a range of things; a potential cycle could be as follows: - Review previous code - Suggest an improvement - Review methods to be changed - Suggest potential impact	At any stage, do not hesitate to ask your partner for clarification about any suggestions that they make, or actions they are working on that you do not necessarily understand.	Think about what your partner is saying and doing. Offering an interpretation of your own understanding of the current state can help move the work forward.	Learning to say I *don't know* or *I don't understand* is critical. Always explain things immediately – try to avoid replying to a question with *you'll see in a while*, as this will distract your partner.	Make a note of previously discussed suggestions and reviews so that similar discussions are not unnecessarily repeated over and over.
Action **Guidelines**	(*for the driver):* Whilst you are programming or thinking about how to structure your code, try to be more verbal – for example, by muttering whilst you are typing. This tends to help the navigator to know that you are actively working, and have a clear sense of how you are approaching the task at hand. If you verbalise your thoughts, this will help the navigator make informed suggestions based on your current actions.	*(for the navigator):* Whilst the driver is programming, actively look to make suggestions that contribute to the code.	*(for the navigator):* If the driver is muttering, use this opportunity to make sure your suggestions have been properly understood.	*(for the driver):* When silent, it can look as if you are clicking randomly on the screen, which risks your navigator becoming bored and distracted. Voicing your thoughts can help counter this.	*(for the navigator):* Think ahead, since you'll be driving in a short while: what is the current course of action not covering? Is there anything worth verifying that might have been left out?

8.4 Suggestions for Future Work

Five aspects of the research described could be investigated further: (i) considering non-verbal communication; (ii) the replication of studies as suggestions for improving on the existing work; (iii) exploring 'lead-in' states for patterns; (iv) widening the target audience and (v) introducing team flow as a way to develop a further understanding of team dynamics within pair programming.

Non-Verbal Communication

This thesis has considered *verbal* intra-pair communication extensively and produced promising results. Some communication exhibited by pair programmers is clearly non-verbal (Sharp and Robinson, 2010, Freudenberg et al., 2007). Non-verbal communication is discussed in Chapter 3 of this thesis, with examples such as a pair member highlighting areas of the screen with the mouse pointer to draw attention; drumming their fingers on the desk when bored or waiting for something to happen; or clearing their throat when disagreeing or trying to draw attention to themselves.

It would be valuable to further explore non-verbal communication: this could provide an additional and alternative dimension to the understanding of pair communication. Once a data-bank of non-verbal communication is built, the interpretation of certain actions could be used to draw inferences, which in turn could be used to derive further guidance for pairs. For example, how should you act if you can see that your navigator is staring out of the window? Does this mean they are bored, and should you re-engage them?

Replication and Further Evaluations

The conclusions stated earlier are derived from the results of several observations and studies. Replication of these studies should be performed to provide further evidence in support of reported conclusions.

There should be two goals: to test for further generalisation of the guidelines against expert pairs (Chapters 3 and 4), and to obtain further feedback and quantitative data from novice pairs (Chapters 5, 6 and 7).

By replicating these studies and providing these guidelines to larger samples from different establishments (both educational, and industry-based), the generalisation and impact of the derived results would be improved. Results obtained by such replications could be used to increase confidence in the general applicability of the guidelines established in this thesis.

Furthermore, evaluations need to be carried out on the additional guidelines informed by comments obtained from industry-based pairs (Chapter 7). As the majority of comments were given by pairs who had been involved in the creation of the original guidelines, it is unclear whether these additional guidelines are, in fact, representative of a wider population. A wider distribution of the survey would lead to more responses from independent pairs, and thus ensure that the next iteration of the guidelines is not subject to any bias.

Exploring ‘Lead-in’ States for Patterns

Most of the analysis described in this thesis investigates how the analytic codes *followed* each other. This gave insight into the conversation flow that the pair

experienced, and gave information about which communication state was most likely to follow. This helped inform the development of the guidelines.

Understanding which codes are most likely to *precede* each other could give information to interpret this conversation flow: this was briefly seen in the sections discussing *Unfocusing* in both Chapters 3 and 4. An understanding of how the states precede each other would not only give a deeper insight into the experience of communication, but would also lead to an understanding of whether or when any states should be avoided.

As looking forwards has informed the development of patterns and guidelines, looking backwards could lead into the development of 'anti-patterns' (Brown et al., 1998): transitions which should be avoided by the pair under certain circumstances. For example, if the pair has not reached their goals and they can be made aware that they are close to an undesirable state (e.g. they are aware that they are close to *Unfocusing*), should this state be actively prevented in favour of more work?

Widening the Target Audience

As seen in Chapter 5, students who were not exposed to the guidelines during the test period expressed frustration at not having had the chance to use them earlier in the semester. The analyses carried out in Chapter 6 (in particular, section 6.4.1.4) illustrates that the guidelines seemed to have a positive effect irrespective of student level; does this mean that the guidelines should be provided to any pair programming student, irrespective of previous experience and study level?

This raises further questions such as: is there a scope for the guidelines to be used beyond academia and education, i.e. in industry? An experienced solo developer is not

necessarily an expert pair programmer. Studies could be designed to measure the long-term impact of exposure to the guidelines to novice pairs in the industry.

Team Flow

Flow is an optimal state of mind, where individuals are so involved in an activity that nothing else seems to matter. The idea represents 'optimal experience', in that when in the flow state, people are so absorbed in their activity that they feel in control of their environment, and experience a skewed sense of time. The experience is not done because it is a compulsory action, but rather, simply for the sake of doing it; when in flow, the individual is so concentrated that they know, moment-by-moment, what their next steps should be (Csikszentmihalyi, 2002). Flow has been applied in areas such as education, including computing (Scherer, 2002, Finneran and Zhang, 2005, Pearce et al., 2005, Bakker, 2005).

Previous research shows that flow-like states have been experienced in pair programming (Belshee, 2005, Lacey, 2006). Sanders (2002) makes the following comment about the students that were observed; "[they] commented on experiencing a skewed perception of time, in which they felt they worked for less time than they actually did".

This raises an interesting question: is it possible for a *pair* to experience this optimal state of mind? If so, how synchronised do the pair need to be with their work and with each other, and is this synchronicity (or flow) expressed through non-verbal gestures? What conditions, both within the pair and in their surrounding environment, could be replicated in order to achieve flow?

8.5 Conclusions

This thesis has investigated the issue of *communication* for inexperienced pair programmers. It has reported on a series of observations of industry expert pair programmers. This work identified communication states frequently experienced by the industry pairs, leading to an understanding of how expert pairs transitioned between various communication states. This knowledge was used to establish communication guidelines for novice pair programmers. Novice pairs reacted positively to the guidelines, indicating that the guidelines were beneficial and useful. Further evaluations indicate that exposure to the guidelines resulted in a positive impact on the students' intra-pair communication, and on their perception of their partner's contribution.

Feedback received from expert pairs was used to add detail to the guidelines, which have been made publically available through a website. At the time of writing, several educators have expressed an interest in adopting the guidelines for the teaching of pair programming within their institutes.

To conclude, this work presents initial evidence showing that it may be possible to improve communication levels between novice students who are pairing together by presenting them with industry-inspired guidelines. Novice pairs who had been exposed to the guidelines reported significant improvements in their perceived communication and partner contribution than students who had not been exposed.

The guidelines developed throughout this thesis can be used to aid pairs that are sceptical or anxious about communicating with a new partner. Novice pairs can use these guidelines to explore different ways of dealing with issues that typically arise during pair programming.

This is captured in the following statement, made by a student participant during the evaluation stages:

> *"There's a definite benefit in introducing this. In pair programming, we're told to 'work in pairs: go!', and there weren't formal steps, apart from the fundamentals. There wasn't a lot of what to do if you became stuck."*

References

ADOLPH, S., HALL, W. & KRUCHTEN, P. 2011. Using grounded theory to study the experience of software development. *Empirical Software Engineering,* 16**,** 487-513.

AIKEN, J. 2004. Technical and human perspectives on pair programming. *SIGSOFT Softw. Eng. Notes,* 29**,** 1-14.

ALLY, M., DARROCH, F. & TOLEMAN, M. 2005. A Framework for Understanding the Factors Influencing Pair Programming Success. *In:* BAUMEISTER, H., MARCHESI, M. & HOLCOMBE, M. (eds.) *Extreme Programming and Agile Processes in Software Engineering.* Springer Berlin / Heidelberg.

ARISHOLM, E., GALLIS, H., DYBA, T. & SJOBERG, D. I. K. 2007. Evaluating Pair Programming with Respect to System Complexity and Programmer Expertise. *Software Engineering, IEEE Transactions on,* 33**,** 65-86.

ATTRIDE-STIRLING, J. 2001. Thematic networks: an analytic tool for qualitative research. *Qualitative research,* 1**,** 385-405.

BAKKER, A. B. 2005. Flow among music teachers and their students: The crossover of peak experiences. *Journal of Vocational Behaviour,* 66**,** 26-44.

BECK, K. 2000. *Extreme programming explained: embrace change*, Addison-Wesley Professional.

BECK, K. 2003. *Test-driven development: by example*, Addison-Wesley Professional.

BECK, K. & ANDRES, C. 2004. *Extreme Programming Explained: Embrace Change (2nd Edition)*, Addison-Wesley Professional.

BEGEL, A. & NAGAPPAN, N. 2008. Pair programming: what's in it for me? Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement, 2008. ACM, 120-128.

BELSHEE, A. Promiscuous pairing and beginner's mind: embrace inexperience [agile programming]. Agile Conference, 2005. Proceedings, 2005. IEEE, 125-131.

BEVAN, J., WERNER, L. & MCDOWELL, C. Guidelines for the use of pair programming in a freshman programming class. Software Engineering Education and Training, 2002.(CSEE&T 2002). Proceedings. 15th Conference on, 2002. IEEE, 100-107.

BRAUGHT, G., EBY, L. M. & WAHLS, T. The effects of pair-programming on individual programming skill. ACM SIGCSE Bulletin, 2008. ACM, 200-204.

BRAUGHT, G., MACCORMICK, J. & WAHLS, T. The benefits of pairing by ability. Proceedings of the 41st ACM technical symposium on Computer science education, 2010. ACM, 249-253.

BROWN, W. H., MALVEAU, R. C. & MOWBRAY, T. J. 1998. AntiPatterns: refactoring software, architectures, and projects in crisis.

BRYANT, S. Double trouble: Mixing qualitative and quantitative methods in the study of extreme programmers. Visual Languages and Human Centric Computing, 2004 IEEE Symposium on, 2004. IEEE, 55-61.

BRYANT, S., ROMERO, P. & DU BOULAY, B. 2006. The Collaborative Nature of Pair Programming. *In:* ABRAHAMSSON, P., MARCHESI, M. & SUCCI, G. (eds.) *Extreme Programming and Agile Processes in Software Engineering.* Springer Berlin/Heidelberg.

BRYMAN, A. 2012. *Social Research Methods*, Oxford University Press.

CAMPIONE, E. & VÉRONIS, J. A large-scale multilingual study of silent pause duration. Speech Prosody 2002, International Conference, 2002.

CHAPARRO, E. A., YUKSEL, A., ROMERO, P. & BRYANT, S. Factors affecting the perceived effectiveness of pair programming in higher education. Proc. PPIG, 2005. Citeseer, 5-18.

CHARMAZ, K. 2006. *Constructing grounded theory: a practical guide through qualitative analysis*, SAGE.

CHI, M. T. 1997. Quantifying qualitative analyses of verbal data: A practical guide. *The journal of the learning sciences*, 6**,** 271-315.

CHMIEL, R. & LOUI, M. C. Debugging: from novice to expert. ACM SIGCSE Bulletin, 2004. ACM, 17-21.

CHOI, K. S., DEEK, F. P. & IM, I. 2009. Pair dynamics in team collaboration. *Computers in Human Behavior*, 25**,** 844-852.

CHONG, J. & HURLBUTT, T. 2007. The Social Dynamics of Pair Programming. *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society.

CHONG, J., PLUMMER, R., LEIFER, L., KLEMMER, S. R., ERIS, O. & TOYE, G. Pair programming: When and why it works. 17th Annual Workshop of the Psychology of Programming Interest Group, 2005 Brighton, UK. 43-48.

CHONG, J. & SIINO, R. 2006. Interruptions on software teams: a comparison of paired and solo programmers. *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work.* Banff, Alberta, Canada: ACM.

CLELAND, S. & MANN, S. Agility in the classroom: Using Agile Development Methods to foster team work and adaptability amongst undergraduate programmers. Proceedings of the 16th Annual NACCQ, 2003. 49-52.

CLIBURN, D. C. 2003. Experiences with pair programming at a small college. *J. Comput. Small Coll.,* 19**,** 20-29.

COCKBURN, A. & WILLIAMS, L. 2001. The costs and benefits of pair programming. *Extreme programming examined.* Addison-Wesley Longman Publishing Co., Inc.

COHEN, D., LINDVALL, M. & COSTA, P. 2004. An introduction to agile methods. *Advances in Computers,* 62**,** 1-66.

COHEN, J. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement,* 20**,** 37-46.

COLEMAN, G. & O'CONNOR, R. 2008. Investigating software process in practice: A grounded theory perspective. *Journal of Systems and Software,* 81**,** 772-784.

CONSTANTINE, L. L. 1995. *Constantine on peopleware,* Yourdon Press Englewood Cliffs.

COPLIEN, J. O. A generative development-process pattern language. Pattern Languages of Program Design, 1995. ACM Press/Addison-Wesley Publishing Co., 183-237.

CORBIN, J. & STRAUSS, A. 2007. *Basics of qualitative research: Techniques and procedures for developing grounded theory*, Sage Publications, Incorporated.

CRABTREE, C. A., SEAMAN, C. B. & NORCIO, A. F. Exploring language in software process elicitation: A grounded theory approach. Proceedings of the 2009 3rd international symposium on empirical software engineering and measurement, 2009. IEEE Computer Society, 324-335.

CSIKSZENTMIHALYI, M. 2002. *Flow: The classic work on how to achieve happiness*, London, Rider.

CUSUMANO, M., MACCORMACK, A., KEMERER, C. F. & CRANDALL, B. 2003. Software development worldwide: The state of the practice. *Software, IEEE,* 20**,** 28-34.

DECLUE, T. H. 2003. Pair programming and pair trading: effects on learning and motivation in a CS2 course. *Journal of Computing Sciences in Colleges,* 18**,** 49-56.

DENZIN, N. K. & LINCOLN, Y. S. 2005. *The Sage handbook of qualitative research*, Sage Publications, Inc.

DI BELLA, E., FRONZA, I., PHAPHOOM, N., SILLITTI, A., SUCCI, G. & VLASENKO, J. 2012. Pair Programming and Software Defects-A Large, Industrial Case Study.

DICK, A. J. & ZARNETT, B. 2002. Paired programming and personality traits. *XP2002, Italy*.

DOMINO, M., COLLINS, R. & HEVNER, A. 2007. Controlled experimentation on adaptations of pair programming. *Information Technology and Management,* 8**,** 297-312.

DREYFUS, H. & DREYFUS, S. 1986. *Mind over Machine: The power of human intuition and expertise in the era of the computer,* Oxford, Wiley-Blackwell.

DYBÅ, T., ARISHOLM, E., SJOBERG, D. I., HANNAY, J. E. & SHULL, F. 2007. Are two heads better than one? On the effectiveness of pair programming. *Software, IEEE,* 24**,** 12-15.

DYBÅ, T., SJØBERG, D. I. & CRUZES, D. S. What works for whom, where, when, and why?: on the role of context in empirical software engineering. Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement, 2012. ACM, 19-28.

FIELD, A. 2009. *Discovering statistics using SPSS,* Sage publications.

FINNERAN, C. & ZHANG, P. 2005. Flow in Computer-Mediated Environments: Promises and Challenges. *Communications of the Association for Information Systems,* 15**,** 82-101.

FLEISS, J. L. 1971. Measuring nominal scale agreement among many raters. *Psychological Bulletin,* 76**,** 378-382.

FLOR, N. V. & HUTCHINS, E. L. A Case Study of Team Programming During Perfective Software Maintenance. 1991. Ablex publishing corporation, 36.

FREUDENBERG, S., ROMERO, P. & DU BOULAY, B. "Talking the talk": Is intermediate-level conversation the key to the pair programming success story? Agile Conference (AGILE), 2007, 13-17 Aug. 2007 2007. 84-91.

FRONZA, I., SILLITTI, A. & SUCCI, G. An interpretation of the results of the analysis of pair programming during novices integration in a team. Proceedings of the 2009 3rd

International Symposium on Empirical Software Engineering and Measurement, 2009. IEEE Computer Society, 225-235.

GALLIS, H., ARISHOLM, E. & DYBA, T. An initial framework for research on pair programming. International Symposium on Empirical Software Engineering, 30 Sept.-1 Oct. 2003 2003. 132-142.

GLASER, B. G. 1978. *Theoretical sensitivity: Advances in the methodology of grounded theory*, Sociology Press Mill Valley, CA.

GLASER, B. G. & STRAUSS, A. L. 1967. *The discovery of grounded theory: Strategies for qualitative research*, Aldine de Gruyter.

GOLD, R. L. 1957. Roles in sociological field observations. *Soc. F.,* 36**,** 217.

GREENE, B. Agile methods applied to embedded firmware development. Agile Development Conference, 2004, 2004. IEEE, 71-77.

HANKS, B. Student attitudes toward pair programming. ACM SIGCSE Bulletin, 2006. ACM, 113-117.

HANKS, B. Becoming Agile using Service Learning in the Software Engineering Course. Agile Conference (AGILE), 2007, 2007. IEEE, 121-127.

HANKS, B., FITZGERALD, S., MCCAULEY, R., MURPHY, L. & ZANDER, C. 2011. Pair programming in education: a literature review. *Computer Science Education,* 21**,** 135-173.

HANNAY, J. E., DYBÅ, T., ARISHOLM, E. & SJØBERG, D. I. 2009. The effectiveness of pair programming: A meta-analysis. *Information and Software Technology,* 51**,** 1110-1122.

HO, C.-W., SLATEN, K., WILLIAMS, L. & BERENSON, S. 2004. Examining the impact of pair programming on female students. *North Carolina State University Department of Computer Science, Raleigh, NC, TR-2004-20*.

HODA, R., NOBLE, J. & MARSHALL, S. 2010. Using grounded theory to study the human aspects of software engineering. *Human Aspects of Software Engineering.* Reno, Nevada: ACM.

HÖFER, A. Video analysis of pair programming. Proceedings of the 2008 international workshop on Scrutinizing agile practices or shoot-out at the agile corral, 2008. ACM, 37-41.

HUGHES, J. & PARKES, S. 2003. Trends in the use of verbal protocol analysis in software engineering research. *Behaviour & Information Technology,* 22**,** 127-140.

HULKKO, H. & ABRAHAMSSON, P. A multiple case study on the impact of pair programming on product quality. Proceedings of the 27th international conference on Software engineering, 2005. ACM, 495-504.

JAEGER, R. G. & HALLIDAY, T. R. 1998. On confirmatory versus exploratory research. *Herpetologica***,** S64-S66.

JENSEN, R. W. 2003. A pair programming experience. *CrossTalk,* 16**,** 22-24.

JOHNSON, D. H. & CARISTI, J. Extreme programming and the software design course. Proceedings of XP Universe, 2001. Citeseer.

JONES, D. L. & FLEMING, S. D. What use is a backseat driver? A qualitative investigation of pair programming. Visual Languages and Human-Centric Computing (VL/HCC), 2013 IEEE Symposium on, 15-19 Sept. 2013 2013. 103-110.

KATIRA, N., WILLIAMS, L. & OSBORNE, J. 2005. Towards increasing the compatibility of student pair programmers. *Proceedings of the 27th international conference on Software engineering.* St. Louis, MO, USA: ACM.

KATIRA, N., WILLIAMS, L., WIEBE, E., MILLER, C., BALIK, S. & GEHRINGER, E. 2004. On understanding compatibility of student pair programmers. *ACM SIGCSE Bulletin,* 36**,** 7-11.

KAVITHA, R. & AHMED, M. I. 2013. Knowledge sharing through pair programming in learning environments: An empirical study. *Education and Information Technologies***,** 1-15.

KINNUNEN, P. & SIMON, B. 2010. Experiencing programming assignments in CS1: the emotional toll. *Proceedings of the Sixth international workshop on Computing education research.* Aarhus, Denmark: ACM.

KINNUNEN, P. & SIMON, B. 2011. CS majors' self-efficacy perceptions in CS1: results in light of social cognitive theory. *Proceedings of the seventh international workshop on Computing education research.* Providence, Rhode Island, USA: ACM.

LACEY, M. Adventures in Promiscuous Pairing: Seeking Beginner's Mind. Agile Conference, 2006, 2006. IEEE, 263-269.

LANDIS, J. R. & KOCH, G. G. 1977. The measurement of observer agreement for categorical data. *biometrics***,** 159-174.

LAZAR, J., FENG, J. H. & HOCHHEISER, H. 2009. *Research methods in human-computer interaction,* Wiley.

LINDVALL, M., BASILI, V. R., BOEHM, B. W., COSTA, P., DANGLE, K., SHULL, F., TESORIERO, R., WILLIAMS, L. A. & ZELKOWITZ, M. V. 2002. Empirical

Findings in Agile Methods. *Proceedings of the Second XP Universe and First Agile Universe Conference on Extreme Programming and Agile Methods - XP/Agile Universe 2002*. Springer-Verlag.

LUCK, G. Subclassing XP: Breaking its rules the right way. Agile Development Conference, 2004, 2004. IEEE, 114-119.

LUI, K. & CHAN, K. 2006. Pair programming productivity: Novice-novice vs. expert-expert. *Int. J. Hum.-Comput. Stud.,* 64**,** 915-925.

MARCANO, A. & PALMER, A. 2009. *pairwith.us* [Online]. Available: http://vimeo.com/channels/pairwithus [Accessed July 31 2014].

MARTIN, A., BIDDLE, R. & NOBLE, J. XP Customer Practices: A Grounded Theory. Agile Conference, 2009. AGILE '09., 24-28 Aug. 2009 2009. 33-40.

MCDOWELL, C., HANKS, B. & WERNER, L. 2003. Experimenting with pair programming in the classroom. *SIGCSE Bull.,* 35**,** 60-64.

MCDOWELL, C., WERNER, L., BULLOCK, H. E. & FERNALD, J. 2006. Pair programming improves student retention, confidence, and program quality. *Communications of the ACM,* 49**,** 90-95.

MCELDUFF, F., CORTINA-BORJA, M., CHAN, S.-K. & WADE, A. 2010. When t-tests or Wilcoxon-Mann-Whitney tests won't do. *Advances in Physiology Education,* 34**,** 128-133.

MELNIK, G. & MAURER, F. 2002. Perceptions of Agile Practices: A Student Survey. *Proceedings of the Second XP Universe and First Agile Universe Conference on Extreme Programming and Agile Methods - XP/Agile Universe 2002*. Springer-Verlag.

MENDES, E., AL-FAKHRI, L. B. & LUXTON-REILLY, A. Investigating pair-programming in a 2 nd-year software development and design computer science course. ACM SIGCSE Bulletin, 2005. ACM, 296-300.

MONTGOMERY, P. & BAILEY, P. H. 2007. Field Notes and Theoretical Memos in Grounded Theory. *Western Journal of Nursing Research,* 29**,** 65-79.

MORRIS, C. W. 1939. *Foundations of the Theory of Signs,* Chicago, University of Chicago Press.

MURPHY, L., FITZGERALD, S., HANKS, B. & MCCAULEY, R. Pair debugging: a transactive discourse analysis. Proceedings of the Sixth international workshop on Computing education research, 2010. ACM, 51-58.

MYERS, M. D. 2008. *Qualitative research in business & management,* London, Sage Publications Ltd.

NAGAPPAN, N., WILLIAMS, L., FERZLI, M., WIEBE, E., YANG, K., MILLER, C. & BALIK, S. Improving the CS1 experience with pair programming. ACM SIGCSE Bulletin, 2003a. ACM, 359-362.

NAGAPPAN, N., WILLIAMS, L., WIEBE, E., MILLER, C., BALIK, S., FERZLI, M. & PETLICK, J. 2003b. Pair learning: With an eye toward future success. *Extreme Programming and Agile Methods-XP/Agile Universe 2003.* Springer.

NAWROCKI, J. & WOJCIECHOWSKI, A. 2001. Experimental evaluation of pair programming. *European Software Control and Metrics (Escom)***,** 99-101.

PANDEY, A., MIKLOS, C., PAUL, M., KAMELI, N., BOUDIGOU, F., VIJAY, V., EAPEN, A., SUTEDJO, I. & MCDERMOTT, W. Application of tightly coupled engineering team for development of test automation software-a real world experience.

Computer Software and Applications Conference, 2003. COMPSAC 2003. Proceedings. 27th Annual International, 2003. IEEE, 56-63.

PEARCE, J. M., AINLEY, M. & HOWARD, S. 2005. The ebb and flow of online learning. *Computers in Human Behavior*, 21**,** 745-771.

PEARSON, J. C., NELSON, P. E., TITSWORTH, S. & HARTER, L. 2006. *Human communication*, McGraw-Hill New York.

PENNINGTON, N. Comprehension strategies in programming. Empirical studies of programmers: second workshop, 1987. Ablex Publishing Corp., 100-113.

PLONKA, L., SEGAL, J., SHARP, H. & VAN DER LINDEN, J. 2011. Collaboration in pair programming: driving and switching. *Agile Processes in Software Engineering and Extreme Programming*. Springer.

PLONKA, L., SHARP, H. & VAN DER LINDEN, J. Disengagement in pair programming: does it matter? Software Engineering (ICSE), 2012 34th International Conference on, 2012. IEEE, 496-506.

PORTER, L., GUZDIAL, M., MCDOWELL, C. & SIMON, B. 2013. Success in introductory programming: what works? *Commun. ACM*, 56**,** 34-36.

PREECE, J., ROGERS, Y. & SHARP, H. 2011. *Interaction Design: Beyond Human-Computer Interaction*, Wiley.

RITCHIE, J. & LEWIS, J. 2003. *Qualitative research practice: a guide for social science students and researchers*, London, Sage Publications Ltd.

ROBSON, C. 2011. *Real World Research*, John Wiley & Sons.

RYU, E. & AGRESTI, A. 2008. Modeling and inference for an ordinal effect size measure. *Statistics in Medicine,* 27**,** 1703-1717.

SALINGER, S., PLONKA, L. & PRECHELT, L. 2008. A coding scheme development methodology using grounded theory for qualitative analysis of pair programming. *Human Technology,* 4**,** 9-25.

SALINGER, S. & PRECHELT, L. What happens during pair programming? Proceedings of the 20th Annual Workshop of the Psychology of Programming Interest Group (PPIG '08), 2008 Lancaster, England.

SALLEH, N. A Systematic Review of Pair Programming Research-Initial Results. Proc. New Zealand Computer Science Research Student Conference (NZCSRSC08), Christchurch, 2008.

SANDERS, D. 2002. Student Perceptions of the Suitability of Extreme and Pair Programming. *In:* MARCHESI, M., SUCCI, G., WELLS, D. & WILLIAMS, L. (eds.) *Extreme Programming Perspectives.* Addison-Wesley Professional.

SANJEK, R. 1990. A vocabulary for fieldnotes. *Fieldnotes: The makings of anthropology***,** 92-121.

SCHERER, M. 2002. Do students care about learning? A conversation with Mihaly Csikszentmihalyi. *Educational Leadership,* 60**,** 12-17.

SELDEN, L. 2005. On Grounded Theory-with some malice. *Journal of Documentation,* 61**,** 114-129.

SFETSOS, P., STAMELOS, I., ANGELIS, L. & DELIGIANNIS, I. 2006. Investigating the Impact of Personality Types on Communication and Collaboration-Viability in Pair

Programming–An Empirical Study. *Extreme Programming and Agile Processes in Software Engineering***,** 43-52.

SHARP, H. & ROBINSON, H. 2010. Three 'C's of agile practice: collaboration, co-ordination and communication. *Agile Software Development.* Springer.

SHERIDAN, V. & STORCH, K. 2009. *Linking the Intercultural and Grounded Theory: Methodological Issues in Migration Research*.

SILLITO, J., MURPHY, G. C. & DE VOLDER, K. Questions programmers ask during software evolution tasks. Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering, 2006. ACM, 23-34.

SIMON, B. & HANKS, B. 2008. First-year students' impressions of pair programming in CS1. *Journal on Educational Resources in Computing (JERIC),* 7**,** 5.

SRIKANTH, H., WILLIAMS, L., WIEBE, E., MILLER, C. & BALIK, S. 2004. On Pair Rotation in the Computer Science Course. *Proceedings of the 17th Conference on Software Engineering Education and Training.* IEEE Computer Society.

STAPEL, K., KNAUSS, E., SCHNEIDER, K. & BECKER, M. 2010. Towards Understanding Communication Structure in Pair Programming. *In:* SILLITTI, A., MARTIN, A., WANG, X. & WHITWORTH, E. (eds.) *Agile Processes in Software Engineering and Extreme Programming.* Springer Berlin Heidelberg.

THOMAS, L., RATCLIFFE, M. & ROBERTSON, A. 2003. Code warriors and code-a-phobes: a study in attitude and pair programming. *SIGCSE Bull.,* 35**,** 363-367.

VANDEGRIFT, T. Coupling pair programming and writing: learning about students' perceptions and processes. Proceedings of the 35th SIGCSE technical symposium on Computer science education, 2004 Norfolk, Virginia, USA. ACM, 2-6.

VANHANEN, J. & KORPI, H. Experiences of using pair programming in an agile project. System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on, 2007. IEEE, 274b-274b.

VANHANEN, J. & LASSENIUS, C. Effects of pair programming at the development team level: an experiment. Empirical Software Engineering, 2005. 2005 International Symposium on, 2005. IEEE, 10 pp.

WATZLAWICK, P., BAVELAS, J. B. & JACKSON, D. D. 1967. Pragmatics of human communication: A study of interactional patterns, pathologies, and paradoxes.

WEBER, R. P. 1990. *Basic content analysis*, Sage Publications, Incorporated.

WERNER, L. L., HANKS, B. & MCDOWELL, C. 2004. Pair-programming helps female computer science students. *Journal on Educational Resources in Computing (JERIC),* 4**,** 4.

WERTZ, F. J., CHARMAZ, K., MCMULLEN, L. M., JOSSELSON, R. & ANDERSON, R. 2011. *Five Ways of Doing Qualitative Analysis: Phenomenological Psychology, Grounded Theory, Discourse Analysis, Narrative Research, and Intuitive Inquiry*, Guilford Press.

WETHERELL, M., TAYLOR, S. & YATES, S. 2001. *Discourse as data: A guide to analysis*, Sage Publications Ltd.

WILLIAMS, L. & KESSLER, R. 2000. All I really need to know about pair programming I learned in kindergarten. *Communications of the ACM,* 43**,** 108-114.

WILLIAMS, L. & KESSLER, R. 2001. Experiments with Industry's "Pair-Programming" Model in the Computer Science Classroom. *Computer Science Education,* 11**,** 7-20.

WILLIAMS, L. & KESSLER, R. 2002. *Pair Programming Illuminated*, Addison-Wesley Longman Publishing Co., Inc.

WILLIAMS, L., KESSLER, R., CUNNINGHAM, W. & JEFFRIES, R. 2000. Strengthening the Case for Pair Programming. *IEEE Software*, 17**,** 19-25.

WILLIAMS, L., LAYMAN, L., OSBORNE, J. & KATIRA, N. Examining the compatibility of student pair programmers. Agile Conference, 2006, 2006. IEEE, 10 pp.-420.

WILLIAMS, L., MCCRICKARD, D. S., LAYMAN, L. & HUSSEIN, K. Eleven guidelines for implementing pair programming in the classroom. Agile, 2008. AGILE'08. Conference, 2008. IEEE, 445-452.

WILLIAMS, L., WIEBE, E., YANG, K., FERZLI, M. & MILLER, C. 2002. In Support of Pair Programming in the Introductory Computer Science Course. *Computer Science Education*, 12**,** 197-212.

WILSON, J. D., HOSKIN, N. & NOSEK, J. T. 1993. The benefits of collaboration for student programmers. *SIGCSE Bull.*, 25**,** 160-164.

WINSLOW, L. E. 1996. Programming pedagogy; a psychological overview. *SIGCSE Bull.*, 28**,** 17-22.

Appendix A: List of *pairwith.us* Videos

The following is a list of the sixty *pairwith.us* videos available on vimeo.com, with reasons given as to why a number of them were rejected from further analysis.

Video Number	Accept	Reasons Why
1	N	Low Quality
2	N	Low Quality
3	N	Low Quality
4	N	Low Quality
5	N	Low Quality
6	N	Low Quality
7	N	Low Quality
8	N	Background noise + external participants
9	N	Background noise + external participants
10	N	Background noise + external participants
11	N	External participants
12	N	A/V issues
13	N	A/V issues
14	N	A/V issues
15	N	A/V issues
16	N	A/V issues
17	N	A/V issues
18	N	A/V issues
19	N	Narrative (not live recording)
20	Y	"Cleaning up"
21	N	Audio/video unsynch
22	N	Audio too low during PP
23	N	Very prominent echo
24	N	Very prominent echo
25	N	Audio/video unsynch
26	Y	
27	Y	
28	Y	
29	Y	
30	Y	
31	N	A/V issues
32	N	A/V issues
33	N	No audio, and lack of video for the last 10 minutes
34	N	Lack of audio for the first 20 minutes (video still continues)
35	Y	
36	Y	
37	Y	
38	Y	
39	Y	
40	Y	
41	Y	
42	Y	
43	Y	
44	Y	
45	Y	Eclipse issues
46	Y	
47	Y	
48	Y	
49	Y	
50	Y	
51	Y	
52	N	Does not exist
53	Y	
54	Y	
55	Y	
56	Y	
57	Y	
58	Y	
59	Y	
60	Y	

Appendix B: Open Coding

This appendix provides a list of memos and scratch notes made during the viewing of selected *pairwith.us* videos as part of the qualitative procedure.

Video #20

- Questions; "where did we leave yesterday?"
- Who runs it? Driver vs navigator?
- One of the pair looks away frequently
- Attention
- Unit tests
- Explanations
- Silence
- Environment; "how do you-"
- Bouncing ideas off each other
- Off-topic chat/banter

Video #26

- Switch-over; "do you know what to do?"
- Discussion about design/logic
- Navigator points out potential issues
- D to N: "what do you think?"
- D gives reasons for what he does
- Coding
- Design + Logic; "how should this work?"
    - No hands on keyboard
    - Quite a long discussion
- Difference of opinion – long-lasting discussion
    - Resolution ~~(annoyance?)~~
    - Who dominates the argument?
    - Driver keeps on trying to "understand" – animated discussion
- "Metaphor disassociation" for the driver
- "We should take a break" even though no resolution reached. Lots of jokes here… to lighten the mood?

Video #27

- Going away seems to have solved the issue.
- Speaking out loudly: "I am doing this ____"
- Driver switch (non-verbal)
    - Method finished; previous driver moves mouse towards new driver
- Code tidying (silent)
- Joke
- Navigator points out issues
- Concentrating/looking at the screen

- Driver seeks confirmation for next stages
- Naming
- "Do you want to drive?"
- Navigator bored?
- Driver stops to "focus": Navigator uses this as cue to speak about expectations and what should happen. This drives the task.
- "I know how to fix this" instigates role switching
- Driver explains to navigator whilst coding
- Both seem to know the project inside out. How would an outsider joining impact this dynamic?
- Silence while coding.
- Muttering under breath.
- Task finished: driver switch.
- Agreement grant (mmhmm)
- Unexpected errors: laughing
- High-five x2
- Constant code review to see var names etc.
- Hand gestures to explain concepts.
- Pop culture references to enforce metaphor
- Navigator: "go for it"
- Discussion about code
    - Naming conventions/metaphors
    - Logic/placement of code
    - Design
    - Refactoring
- Navigator explains error/issue
- Driver: "I'm in a happy place now!"
- Keyboard controls/shortcuts issues
- Code/Refactoring
- Environment
- Confirmation from each other
- Navigator prompts where to look on screen

Video #28

- "What we did, what we're doing"
- "When you've done that can I just borrow it (drive) for a second?"
- Coding
- Replicating errors
- "Whose turn is it to drive?"
- Silence
- Driver asks about method name
- Driver stops mid-sentence; navigator: "hmm? Yeah?"
- Importance/significance of code
- Distraction – both ignore it until navigator looks and "ooh"s at which point the driver stops working and also gets distracted. Driver gets back into it, but the navigator is still distracted.
- Navigator points out potential issues.
- Driver: "hmm" / Navigator: "you didn't import"

- Silence; navigator only replies to driver prompts
- Both not sure of how to solve problem. Navigator offers step-by-step suggestions which fail. Driver offers own input.
- Navigator breaks concentration, drinks, stretches, looks around… trying to help.
- "Can I just take over?"
- Navigator knows what to do and wants to do it.
- Driver asks for clarification.
- Driver shows "different ways" of achieving something.
- More high-fiving.
- Navigator: "We talked about this before"
- Discussing behaviour of program.
- Navigator reads whilst driver highlights and points with the mouse.
- Navigator dominates discussion and temporarily becomes driver.
- Navigator narrates and driver types code.
- Driver: "mhmm"

Video #29

- Can't remember/"let's work it out together"
- Logically stepping through lines of code
- Driver asks navigator to repeat advice.
- Laughter at something the driver typed
    - (Is coding a type of communication?)
- Navigator prompts driver and makes suggestions
- Phone beeps – both look and return to code
- Discuss errors and warnings
- Driver: "we need to _____"
- Discuss functions of code
- Discussion over whether code is duplicate
- Mouse used to read code by driver for navigator
- Navigator dictates code/explains logic
- "That's not right!" / "That's what I meant!"
- Driver types with no prompting
- Bouncing ideas
- Phone rings;
    - Driver: "Is that yours?"
    - Navigator: "Yes" and ignores
- Phone rings again – navigator leaves the room
- After navigator leaves, the driver is <u>still</u> narrating what he is doing.
- Does not continue AFTER the pre-assigned task until navigator returns
- Repository
- Discussion on reverting changes
- Planning next steps/going over previous work
- Navigator: "I would prefer it if you actually were to run it again"
- Joke whilst waiting for compilation
- Ice-cream van (interruption)

Video #30

- Driver reads code and waits for navigator to approve

- Navigator says what needs to be done – driver follows orders
- Navigator offers a choice; how do you want to do this?
- Switch: “Can I give an example?”
- Navigator: “What’s the advantage of doing it this way?” / Driver: “None” and deletes changes
- Refactoring
- Talk about code placement
- Reads out code when typing
- Discuss merits of copy/paste code
- Silence
- Coding
- Switch: “I think… can I have a go again?”
- Refactoring and discussing possibilities
- Switch: “Let’s do it like this.”
- Switch: “Over to you.”
- Joke: “You are our only hope”
- Using code as prompts: [(types) / “Yeah” / “Mm” / “Ok”]
- Code explanation (proving the need)
- Discussing possibilities for code
- Explaining errors
- Code aesthetics (“would have had it on one line”)
- Environment shortcuts
- “Back to here, and it’s going to be…”
- High-five
- Tongue-clicking / “Enter”
- Switch: “Can I just drive for a second?”
- Code rationale
- “This is wrong” / “Why?” / “Because _____”
- Code aesthetics/prompt by code
- Joke: “I am ready to commit. No commitment issues!”
- Repository and discussion of text
- Error: discussing why it doesn’t work

Video #35

- Off-topic discussion
- Choosing next task
- Coding
- Discussing expected behaviour
- Silence while coding
- Firefox updates distract/stall progress
- Navigator finds an error
- Switch: “Can I just drive for one second?”
- “I’ve lost interest in this feature”
    - This prompts a logic/code review
- Refactoring
- Off-topic chat (urge to sneeze)
- Window not closing – PP unsure why
- Navigator dictating what driver does.

Video #36

- "I can't remember what we're doing"
- "I want to drive"
- Driver does not agree with what it being coded – navigator convinces him
- Discussion of logic
- Joke
- This is what we will do and what we expect to happen/break/fail
- Discuss errors (good/bad)
- "Ah, that's interesting"
- Planning: "do we need to do that now?" / "No, let's leave it for a minute"
- Joke when naming
- Navigator makes a suggestion
- Driver working on tricky code
- Coding
- "Why is this broken?"
- Navigator: "We are getting de-sensitized to things. I don't like it."
- High-five upon completion

Video #37

- Review previous work
- Refactoring
- Pointing to screen
- Navigator offers suggestions; "I would like this to be in a method"
- High-give.
- Switch: "can I run the next tests?"
- Talk about naming classes/methods and metaphors
- New method: PPS discuss what it should do and say whilst working on it
- Coding and narration
- Argument
- Naming and structure
- Environment
- Navigator points out a mistake in typing.
- "Can I make a suggestion?"
- Coding and naming
- Silence

Video #38

- Coding exceptions (and narrating)
- Using tooltips to think about exceptions
- Navigator introduces what should be done
- Navigator sits back (behind) driver
- Discuss logic/what things should do
- Sequence discussion
- Switch when task finishes
- High-five
- "Stating the obvious" – does not contribute to work
- Silence while coding

- Discussing environment/IDE
- "We should take a short break" / "We're going to miss obvious answers"

Video #39

- Narrate why error happens
- "Should we ___?" prompts instruction from driver to navigator
- Use warnings and errors to create future tasks
- Discuss code theory (generics, static, etc…)
- High-five
- Driver hands keyboard over
- Navigator dictates next steps
- Approval
- Father's Day
- Metaphor
- Joke about naming things
- Bouncing ideas off each other
- Reassurance
- Repository
- Navigator realises why something "failed"
- Navigator gives ideas
- Next steps decided – taking a break.

Video #40

- Plans for this session
- Talk about what needs to be done
- "I've got a good idea" / (grunt of approval)
- "It should…" → expectation
- Explains what needs to be done
- "What has to be done?"
- Logic and code structure
- Aesthetics
- Use of this/here
- Trying to find a natural place to stop

Video #41

- Recall "what we did" from the last session
- Environment
- Using failing tests as prompts
- Navigator takes control of mouse to point and emphasize a point
- Navigator gives implicit instructions and help:
    - N: "Import it"
    - D: "It's not liking it"
    - N: "Did you import it?" → repeated
    - D: "Yeah"
- Not sure how to solve an error. Some 'pointless' comments; e.g. "It should be resolved!" or "Why is it doing what?" eventually prompts suggestions.
- "There we go"/"Yay"/High-five
- Finding solutions

- Switch; “I’ll show you what I mean”
- Silence whilst considering repercussions
- “What should the next step be?”
- Switch: “It’s your turn to write a test”
- Unit tests
- Ask question about existing methods
- Figuring out code logic
- Naming objects
- Stating “the obvious”
- Frustration (“arrrrrgh”) at key-press, scoffs
- Navigator: “Mmhmm”
- “Must be your go”
- Muttering whilst typing prompts future steps
- Navigator prompts next steps
- Navigator predicts results from test and compiler
- Jokes

Video #42

- Joking
- Naming conventions
- Writing tests
- Comment: AP had commented that he was navigating too much, so AM says he can drive now.
- Reassurance (navigator → driver)
- Answering questions
- Navigator apologises for being tired. This leads to off-topic talk about hay-fever.
- Narrating while coding
- Repository/”We should commit”
- Very loud helicopter leads to disruptions and jokes
- Navigator makes a request to refactor
- Talking about the weather
- Code readability
- Navigator explains next possible steps
    - Driver gives alternatives
- Driver anticipates problems and solutions
- Navigator suggests renaming method
- Future-proofing code

Video #43

- While not working, navigator spots inconsistency
- Refactoring
- Next steps planned and agreed
- Navigator: “actually, what I’d like to do…”
- Acceptance test error
- Tired (both comment)
- Looking for a problem: both reading various parts of the code that seem wrong and try to work it out logically.
- Navigator wants to test before re-breaking.

- Switch: “You’ve done a lot of driving”
- Joke/pun re: driving
- Navigator corrects previous code and questions driver.
- Driver is asking permission to do this.
- Discussion on speed of mouse
- Navigator leaves to fix lights; “the buzzing is annoying me”
- Pun/laughter
- Error: (“uh-oh”) causes PP to stop early for a coffee break

Video #44

- Use of mouse pointer and “that”
- Muttering/speaking while coding
- Driver seems focused. Explaining things to navigator who seems to clue in quickly and offer pointers.
- Driver runs ideas by navigator
- Switch: “I’m not clear what you mean. You drive, you drive – show me what you mean”
- Navigator: “Can I just interrupt for a second?”
- Design vs behaviour
- “What did we do the other day?”
- “I’m convinced it doesn’t belong there.”
- Muttering/live narration
- Bouncing ideas off each other
- Planning next stages
- Problems with the IDE
- Code aesthetic/refactoring/logic
- Reading errors
- Driver: “did you delete this?”
- Driver finds solution and explains it to navigator
- Repository: found stuff that should not exist.
- “Let’s F5 it!” / “F5 what?” / Clarification
- Getting annoyed/louder
- Use each other for reassurance
- Navigator attempts to set ‘goals’ – “it’s late, we’re tired – just get it back to how it was.”
- Driver ‘ignoring’ navigator to fix error – navigator clears throat to get driver’s attention.
- Unit tests
- Stop due to tiredness.

Video #45

- Review of previous code
- Silence
- Planning ahead
- Figuring things out whilst discussing
- Navigator points out naming issue
- Driver works, navigator “mmhmm”ing along

- Using Windows keyboard on a Mac seems to cause keypress problems and disruptions
- Narration whilst coding
- New problem found – navigator is annoyed
- Driver encounters errors – navigator points to solution
- High-five.
- "Next thing?"
- Joke
- Keyboard shortcuts provided by navigator
- Joke

Video #46

- "I want you to start driving"
- Code whilst narrating
- Navigator: "Dot dot" / "Why?" / Explains
- Grammar discussion
- Joke
- Talk about past experiences of pair coding
- Banter; relaxed
- Discuss what should happen
- Finishing each other's sentences
- Joking
- Trying to debug
- What should happen to the code
- High-five
- Navigator: "Can I see ___ again? One thing I didn't like…"

Video #47

- Go over new code, pointing out areas that need to be improved
- Suggestion by driver (defending)
- Navigator agrees but driver pushes suggestion defensively. Navigator: "cool"
- Anticipating next step whilst coding
- Reading code → logic
- Driver deletes, navigator asks for 'undo'
- Navigator concerned about adding additional layers to the code
- Figuring out logic
- Switch denied: "Do you want me to do this?" / "It doesn't make much difference"
- Code is refactored/useless code removed
- Code logic
- Switch: "Can I drive?"
- Brainstorming.
- Looking at code in silence
- Driver: "how do I do this thing?" prompts navigator to offer various solutions
- Discussion about possible outcomes
- Pre-empting problems
- Reading code to make sense of it
- Decide to take a break

Video #48

- Plan next stages
- Code review
- Code structure/logic/refactoring
- Driver speaks whilst typing – navigator prompts ideas when there is a pause
- After disagreement on how to code, role switch to clean things up
- Navigator uses pen and paper to figure things out
- Brainstorming; "what happens if we get rid of ____?"
- "Are we going off-track now?"
- Navigator references a blog post to discuss a possible way of coding/implementation
- Removing redundant tests
- Failing tests – discuss individually

Video #49

- Indicating problems from previous session
- Possible solutions are discussed.
- Phone buzzes – does not cause distraction.
- Driver keeps on coding and talking whilst navigator does miscellaneous things (opening window, etc…)
- Coding.
- Confirmation sought ("did I just…?" / "yeah")
- Rhetorical questions: "why is that not working?"
- Speaking whilst typing
- Determining logic
- Environment discussion on keyboard shortcuts
- Navigator dictates code to driver
- Using code to 'prove' what they are thinking
- "I'll drive to show you"
- Use failing tests as prompts
- High-fiving when success is achieved.

Video #50

- Uncompleted items from previous session
- Explaining backstory – re-examining logic?
- Coding and speaking whilst typing
- Discussing assertion test errors
- Joking
- Off-topic discussing
- "Do you think we should-?"

Video #51

- Explain previous work
- Navigator picking up on driver's quirks (deleting brackets from the 'other side')
- "What have we actually changed?"
- Code structure discussion
- Different possibilities

- Navigator explains how to do next bit of code to driver
- Clarifying packages and class locations
- Next steps discussed

Video #53

- What needs to be done (and why) discussed
- Joking
- Driver explains his vision to navigator and gets him to agree to this way of doing this
- Code structure
- Navigator prompts driver
- Phone interrupts both; "do you want to answer"; driver reels back the conversation.
- Coding
- Phone (again) put on silent but interrupts both
- Navigator gets back on topic.
- Time awareness

Video #54

- "I've been doing loads – your turn to drive!"
- Navigator points out discrepancy.
- Driver: "trust me – I know what I'm doing"
- Off-topic chat; Windows vs Mac keyboards…
- Code
- Errors to prompt tasks
- Navigator offers advice
- Coding and naming
- High-fiving

Video #55

- D: "This seems all right" / N: "Show me on the RHS again?" / D: Why are we using a list?"
- Explaining next steps
- Code logic/structure (navigator is making suggestions to the driver)
- Noise in background but PP not reacting
- A Twitter app starts to create loads of popups, distracting both programmers
- D confirms actions with N before proceeding
- Coffee break – banter
- "Move that method up so it makes sense"
- Driver talks about coding plans, but navigator interrupts – prevents errors?
- Navigator offers instruction: "now you can delete"
- Discussion about refactoring (functional vs pretty code)
- Use of analogies and metaphors to explain point
- Planning future steps (refactor, or fail the next test?)
- Navigator suggests running an acceptance test

Video #56

- Off-camera conversation helps clarify the disagreement between aesthetic and functional code
- Phone rings – no reaction
- Dictating while coding to prompt next steps
- High-five
- Read out method names in a class and considers changing method order
- Next steps suggested by navigator
- Coding
- Reading code helps understanding of logic
- Driver looks at navigator to agree before proceeding
- Driver reassures navigator on some concerns
- Navigator points out missing code to the driver
- Driver dictating with navigator occasionally prompting
- Next tasks planned

Video #57

- Fix failing tests from previous sessions
- Dictating while coding
- D: "I'm not sure what I've done" / N: "You've _____"
- Noise disrupts PPs
- N: "This is too big a step"
- Fixing errors
- Navigator suggests solutions when driver falters
- Navigator guiding driver
- High-five when success is achieved
- Navigator offers constant encouragement ("excellent!"/"last one…")
- N: "Still a bit 'meh' – but it's getting there"
- Looking back at previous code and learning from mistakes

Video #58

- Discussing fonts
- Moving methods
- Discussing what code does and logic behind next actions
- Driver comes up with idea and proposes action/navigator agrees
- Discussing responsibilities of the class
- Switch to explain next steps
- Test fixing
- Navigator finds a problem
- Brainstorm (different ways to solve one issue)
- Discussion about keyboard shortcut

Video #59

- Writing classes
- Navigator makes several suggestions
- Navigator looks for possible fixes
- Worried about time constraints
- Driver breaks task down and types/dictates

- Both prompt each other while driver types
- Method placement
- Switch: “This is probably a good time to hand over to you”

Video #60

- “Where were we?”
- Next steps explained
- “Can I just interrupt you?”
- Navigator explains why driver is wrong
- Code logic re-evaluated
- High-five
- “It’s good to be through this”
- Driver explains next steps
- Acceptance tests discussed/refactoring occurs
- Discussing ways of improving methods
- “I’m happy with that”
- Use warnings as prompts (“where to start next--”).

Appendix C: Transcripts

This appendix provides one of the *pairwith.us* transcripts that was created and used for qualitative analysis. A copy of the remaining the transcripts (unformatted) is available online[13].

***pairwith.us* video #20**

A	B
Whilst I was away I had a thought on how to do it. That's pretty much where we left off. Even though we wrote the tests... it was a learning experience.	
	The next thing is the acceptance tests. We have our tasks list here.
Oh! Ah. I hadn't seen that.	
	Let's start the pomodoro. First of all the first step is to 'ignore' things that they are not interested in.
Hopefully that's the case of doing Team-	
	I don't think there's an ignore, um, on the… oh there is
RecentChanges?	
	Yeah. Unless you really want that. D'you know - I wish I could switch RecentChanges off.
Yeah, that would be the-	
	Cos if I want to know what's changed I would look in Subversion.
Yep. That's something maybe for… for a future uh enhancement.	
	A project you can do.
Well I think it's directly related to Fitnesse but, um, Fitnesse is very active at the moment so, uh, it might be that RecentChanges isn't relevant anymore. I think more and more people are actually um checking their things into Subversion so it might be that these things just...	
	Do you know what we should do? We should change RecentChanges content so that it says um cos we can't actually disable RecentChanges so we should put some content on there saying um in order to find out about

[13] All transcripts can be found on Dropbox at the following link: https://db.tt/YiOstuMF

RecentChanges look in the, uh…

Subversion logs.

Well, Mercurial logs.

Ah yeah. Uh. Did you say subvert-

I did!... Say subversion. I meant Mercurial.

No, we didn't mean it.
Well it's, um, the recent changes, um.... I think, I think cos there is, there is already something in there that's not in this, so we need to delete it first.

It’s modified. If we cancel that. Stop Fitnesse.

We don't want it running - delete - it gets very upset. We don't want to see it very upset!

You wouldn't like me when I'm angry!

Two different meanings there. Your one's the Hulk. Do you know what mine was from? ...'You haven't seen me... very upset' No?

I might do but I don't know.

Mission Impossible. The first one, yeah.

In the cafe... with... whatever, after his bosses apparently just died.

Yep, delete.

Delete.

Delete. Like the Cybermen man, like the Cybermen.

And then we can commit that.

Yep.

Get rid of those...
Yeah?

Yep.

Two files that have been wrongly modified by recent repository changes.

Revert them to their previous versions.

There's no discernible difference.

Might just be that we went into the edit view and saved it.
That can be reverted.

Refresh the project.
Plugin doesn't seem to...

Switch settings? F5?

Why is it still saying we've got changes then?
Huh. No files.
Alright, that's, that's-

Can you quit, um, Eclipse and start it up again?

Shall I cancel that?

The reason was so that can refresh the packages.
Before you do that, create the files again.

I shouldn't need to ignore them. That one?

Excellent.

Do I refresh the project?
What were we ignoring?

RecentChanges.

Change the technical stuff. Change that to- um.
Something else like we want it to be useful.

I'll bring, um, what is it doing?

This guy is providing the language.

Yeah.

Like… an interpreter.

Hmm.

Define interpreter for the rest of the page?

Just for page?
So we've got RecentChanges and we haven't any errors yet, so we'll run the test and see what happens.

It might still be running.

I think... different process.

You're going to activitymonitor again?
Can we not... next task?

We've got error logs but that's progress cuz they're ignored.

That's good.
I'm feeling it.

We got recentchanges, result application...

Can you delete it?

See removed? It's committed.

Just the fitnesse jar…

We need that. There?
We should probably…

Make it so it copies that fitnesse across.
That's not important right now.

Ok so the last problem we solved.

Yeah except it's not bringing back the jar

Yeah.

That’s exactly the same..

Mmm.

We didn't actually commit the...

I got it back.

Tidying up the um-

Right, ok, so we want activity monitor.
Off.

Cool.

D'you know what? I personally think that that’s a small value including that right now. Why I can't I see... and then we want Java.

Let's run the acceptance tests now.

Go see if you can go to the-

Yeah.

Now let's try running those acceptance tests again haha!

Ok so now we're back to missing an event um but we still have that leftover.

We need to kill the-

Have you refreshed?

I think it's because it's in the ignore
Can we edit the htignore?
That's probably what's causing the problems.

Should we try to do this manually?

Go view history. Show history. Revert back to a specific change.

Update, yeah that should do it
So you just deleted something that we thought we didn't need.

Oh you got it back. Good.

Hgignore.

...Lists of things to do sort out, um…

Automatically...

Shall we go have a look?

Haha.. I think we killed the wrong one!

-See if you can go to the other one.
It is running.

Just not very helpful.

Cool.

Ok. So what I'd like to do is write the part of the acceptance test that would force us to implement the killing the wiki process every

Ok.

I think-
We would need to know about all of our actors; what we can do and why.

time automatically when we run it? And then I'd like to supply the code for you there.

So should we edit the page? If you just put-

Appendix D: Other Examples of the Coding Scheme

The coding scheme used in this thesis is defined, using exemplars, in Chapter 3. Further examples of conversation fragments used for each code are shown below.

Review

N: We were – we were looking at – at the end of the pomodoro, at the end of the day we had some red lines.

D: What are we doing there?

D: We did the Librarian, fixed that code.

N: The tests?

D: And that test is all done.

N: We've got ten warnings.

In the two cases above, the pair is reviewing code that had been written during the previous pairing session.

The first exemplar shows the pair reminding each other where they had left off at the end of the previous session. The second exemplar occurs following some explanatory chat. It can be seen that they are reviewing finished classes and tests related to a Librarian function in the code. Review ends once the navigator spotted ten warnings, at which point the driver and navigator start to suggest possible courses of action.

D: The Expert is working with the Librarian and the Librarian is looking good.

N: It's looking very good. The other one – CastingAgent – it's looking good.

D: Director's looking good.

N: I'm happy with that now.

D: Me too.

This transcript gives a conversational fragment that occurred at the end of one of the pair's sessions. They are reviewing completed code that had been under development for the past week. This comes across as a box-ticking exercise – the pair is going through the new parts of the code that were added to the system, and confirming that they work as expected.

Suggesting

N: Give it a hash map, and a fake actor.

D: We need to put a new hash map there. When we create a dressingRoom here, we need to move that one there.

D: We could possibly use – um –

N: The process idea from runtime. Maybe we should do something like generate a new port number.

The pair, in these cases, is suggesting different ways of fixing the problem at hand. The first transcript shows both the navigator and the driver making suggestions about how to write the code, whereas the latter transcript shows the navigator making a suggestion, prompted by the driver's uncertainty.

Explanation

> *D: What do you mean, you know why it's crashing?*
>
> *N: The reason was to refresh the packages so now there's no warnings or changes. That's the therapistCannotHelp error – so that means – this is the wrong type of therapist.*

This instance shows a member of the pair explaining in some detail *why* the code is not functioning as expected. In these particular cases, the explanation comes as a reaction to a member of the pair voicing confusion, or surprise.

> *D: The abstraction is not necessarily the CastingAgent, it's because the Librarian… it's doing it in a specialised way, but now it's not so specialised. It's… oh, it's a cast. So the CastingDirector casts the character… keeps a hold of him… so we can keep the Librarian and find the demographic.*

In the full transcript, this explanation comes at the end of a Review of legacy code, where the pair is discussing the need for the current piece of code. This

instance shows the driver justifying its existence by explaining the thought process and rationale behind writing the code in question.

Code Discussion

D: I think it's a good example of the level of feedback and the cycle time.

N: We want to use acceptance tests in this way.

D: This is much more the sort of level we work at.

N: I didn't realise we were this close!

These instances show the pair (or a member of the pair) making comments about the code, its underlying structure, and about the compiler's interpretation of the code. This type of commenting occurs quite frequently in the observed videos.

N: How did you do that?

D: Tab completion.

N: I thought that-

D: -it does work. Tab completion does work.

N: It shouldn't – I just want to see it for myself.

In this case, these transcripts were followed by suggestions made on alternative shortcuts that could be used, or how the discovered features could be used to solve the problem at hand.

Muttering

D: This type of thing… arrays… dot… as… import… that should do it.

D: Um… error… logs… uh… and… example application.

N: Yeah.

In these cases, the driver is muttering about the code as he is typing it down. The navigator in both instances is looking intently at the screen. Following the first transcript, an error is spotted, and therefore the navigator makes a suggestion on how to fix that.

Unfocusing

N: My son – we're going for a meal tonight. It's Father's Day.

D: That's nice. Did you get anything?

N: He got me a card. It says 'number one dad'.

D: …Number one dad? That just makes me think - who's his number two dad?

The pair is discussing each other's Father's Day plans, which prompts one of them to make a joke about a card received. The conversation is entirely off-topic and occurred whilst waiting for their code to compile.

D: 'You haven't seen me very upset'.

N: "Mission Impossible"? The first one. Tom Cruise in the café.

D: After his boss has apparently just died.

This instance of unfocusing occurred in response to several errors appearing on the screen. A member of the pair directly quotes the film 'Mission Impossible'. His partner notices this fact, and a short conversation about the context of the quote follows. Following this, the pair start re-reading the code to start debugging.

Appendix E: Observations within Industry

The following sections consist of consent forms, information sheets and surveys that were used during all evaluations within this thesis. Ethical approval was obtained prior to each study from the University of Dundee SoC Ethics Board.

E1 Forms used for Evaluations with Industry Members

School of Computing

Information Sheet

The Issue

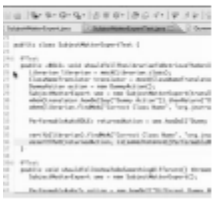
- There is evidence to support the claim that pair programming has certain benefits over more traditional "solo" coding.
- Pilot experiments have shown that 50% of pair programming failures have been attributed to "communication" reasons.
- Little work has been done to study intra-pair communication

Research Proposal

- I am planning to investigate how experienced pairs communicate within their pairs, whilst programming. Through video recordings, conversation analysis and interviews, I'm seeking to gain an insight in communication topics and trends.

Outcomes

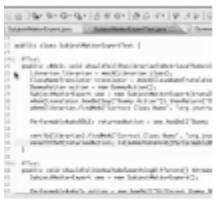
- This research can help with:
  - Improving the understanding the scientific community has of communication within pairs
  - Understanding the various roles within the pair
  - Communication guidelines for novices and remote programmers



If you have any further questions, please contact me: (markzarb@computing.dundee.ac.uk). Thank you for your time.

Mark Zarb is a PhD student at the University of Dundee, supervised by Dr Janet Hughes and Prof John Richards.

Your Involvement



Images from the pilot investigation at the University of Dundee

Thank you for considering helping with this research into further understanding interactions within pairs. This page details the steps required for participation.

Beforehand

- You are asked to give written consent to the project, lasting the entirety of the observation phase.
  - Participation is completely voluntary. You are asked to give your consent to be observed, and have your sessions recorded. Please note that you may withdraw from the research from any time without any penalty, and without stating a reason, with no repercussions. Any data collected shall be treated with full confidentiality, and if published, will not be identifiable as yours. All recordings are not identifiable unless prior written permission is given by you. No undue risk arises from participation in this study.
- We arrange a time and duration for the data collection session(s).
  - This will be pre-arranged.

Observations

- A camcorder/webcam/dictaphone is put in place, to start the recording.
- The participants work as usual for the duration of the session.
  - The observer (myself) shall not be present during the recording of the sessions, for minimal disruption.
- The participants complete a questionnaire.
  - This is a short questionnaire on their background as software developers, their working relationship within the pair, and their satisfaction with the pairing session. All questions are completely optional.

Follow up

- I can provide a debrief detailing any results arising from this data.

If you have any further questions, please contact me: [markzarb@computing.dundee.ac.uk]. Thank you for your time.

School of Computing

Consent Form

Please read the previous pages carefully, and make sure you understand the procedure. If you are unsure about any of the information presented, feel free to ask the module tutor or your lecturer.

Please note that any observations or footage gathered from these sessions shall only be used for research purposes.

Images from the pilot investigation at the University of Dundee

Optional Consent

This does not affect your eligibility for this study.

• I consent to pictures being taken for research purposes	Yes	No
• I consent to video being taken for research purposes	Yes	No
• I consent to audio being recorded for research purposes	Yes	No
• I consent to participate in follow-up interviews	Yes	No

Consent

If you agree to participate, please complete this section

• I agree to the above conditions, and am willing to participate in this study.	Yes	No

- Name: _____
- Signature: _____
- Researcher's Signature: _____
- Date: _____

E2 Survey used with Industry Members

End-of-session Questionnaire

Any data from this questionnaire used in presentations or other publications will be kept anonymous.

Instructions:	*Section A:* *Please ensure that you read each statement carefully. Complete the items listed overleaf by rating the statement along the scale.*

Please note:

Please answer questions as fully as possible. All answers given are confidential and will not be disclosed to anyone other than authorised researchers at the University of Dundee.

You do not need to answer any or all questions contained in this questionnaire.

1. This session was typical of a standard pair programming session.

Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree

2. I feel pair programming is more beneficial than *solo* programming.

Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree

3. During this session, I found communicating with my partner to be easy.

Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree

4. How long have you been a professional *solo* programmer?

5. How long have you been pair programming (with other people)?

6. How long have you been pair programming (with today's partner)?

Appendix F: Observations with Students

F1 Forms used for Observations with Students

School of Computing — DUNDEE

Information Sheet

The Issue

- Pair programming has been proven to have certain benefits over more traditional "solo" coding.
- Pilot experiments have shown that 50% of pair programming failures have been attributed to "communication" reasons.
- Little work has been done to study intra-pair communication

Research Proposal

- I am planning to investigate how experts communicate within their pairs, whilst programming. Through video recordings, conversation analysis and interviews, I'm seeking to gain an insight in communication topics and trends.
- Following an initial observation phase, you will be taught various tips and guidelines that aim to enhance your pair experience, such as ways to keep the communication active, and ways to "navigate" or "drive" more effectively. You may then be asked to incorporate these techniques in the way you work.

Outcomes

- This research can help with:
  - Improving the understanding the scientific community has of communication within pairs
  - Understanding the various roles within the pair
  - Communication guidelines for novices and remote programmers

Images from the pilot investigation at the University of Dundee

If you have any further questions, please contact me: (markzarb@computing.dundee.ac.uk). Thank you for your time.

Mark Zarb is a PhD student at the University of Dundee, supervised by Dr Janet Hughes and Prof John Richards.

School of Computing

Your Involvement

Images from the pilot investigation at the University of Dundee

Thank you for considering helping with this research into further understanding interactions within pairs. This page details the steps required for participation.

Beforehand

- You are asked to give written consent to the project, lasting the entirety of the module.
  - Participation is completely voluntary. You are asked to give your consent to be observed, and optionally, have your sessions recorded. Please note that you may withdraw from the research from any time without any penalty, and without stating a reason, with no repercussions. Any data collected shall be treated with full confidentiality, and if published, will not be identifiable as yours. All recordings are not identifiable unless prior written permission is given by you. No undue risk arises from participation in this study.
- We arrange a time and duration for the data collection session(s).
  - This should normally happen during the allocated lab time for the module.

Observations

- A camcorder/webcam/dictaphone is put in place, to start the recording.
- The participants work as usual for the duration of the session.
  - Following an initial 3-week period, you may be asked to try and incorporate some tips and techniques in the way you work and communicate with your partner.
- The participants complete a questionnaire.
  - This is a short questionnaire on their background as software developers, their working relationship within the pair, and their satisfaction with the pairing session. All questions are completely optional.

Follow up

- I can provide a debrief detailing any results arising from this data.
- You may be asked to attend an informal interview or a focus group.

If you have any further questions, please contact me: [markzarb@computing.dundee.ac.uk]. Thank you for your time.

School of Computing

Consent Form

Images from the pilot investigation at the University of Dundee

Please read the previous pages carefully, and make sure you understand the procedure. If you are unsure about any of the information presented, feel free to ask the module tutor or your lecturer.

Please note that any observations or footage gathered from these sessions shall only be used for research purposes and will not be viewed by anyone involved in this project other than the researcher until your grades are released. Your behaviour will in no way impact your grades.

Optional Consent

This does not affect your eligibility for this study.

• I consent to pictures being taken for research purposes	Yes	No
• I consent to video being taken for research purposes	Yes	No
• I consent to audio being recorded for research purposes	Yes	No
• I consent to participate in follow-up interviews	Yes	No

Consent

If you agree to participate, please complete this section

• I agree to the above conditions, and am willing to participate in this study.	Yes	No

- Name: _____
- Signature: _____
- Researcher's Signature: _____
- Date: _____

F2 Survey used with Observed Students

End-of-Session Survey

Participant ID*: _____
*(leave blank)

Any data from this questionnaire used in presentations or other publications will be kept anonymous.

Instructions: *Please ensure that you read each statement carefully. Complete the items listed below by rating the statement along the scale.*

Instructions:

Please answer questions as fully as possible. All answers given are confidential and will not be disclosed to anyone other than authorised researchers at the University of Dundee. Information will **not** be shared with other staff in the research project until all grades are released. The researcher has no input in any grades.

You do not need to answer any or all questions contained in this questionnaire.

Please ensure you fill one of these sheets once per week (sheets can be obtained from your lab tutor).

On Friday, your team leader is to collect all your team's sheets and put them in the red box marked Pod 2.05 on the first floor of the QMB.

Your name: _____

Who did you pair with? _____
(if you paired with more than one person during this session, please fill in a separate sheet for each partner).

Please fill one of these sheets individually after every lab session for this module.

1. This session was enjoyable (in terms of pair programming).

Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree

2. I feel pair programming is more beneficial than *solo* programming.

Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree

3. During this session, I found communicating with my partner to be easy.

Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree

4. There were no periods of uncomfortable silence during this session.

Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree

5. I was confident (not anxious) during this session.

Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree

6. What percentage of your work (design, analysis, coding) was done in pairs?

0 – 20%	21 – 40%	41 – 60%	61 – 80%	81 – 100%

7. How many hours this week have you spent pair programming?

8. Rate your partner's contribution to today's session.

No Participation	Marginal	Satisfactory	Very Good	Excellent

F3 Semi-Structured Interview Protocol

Phase 1

- What were your expectations of pair programming before you started using this technique?

- Did your experience meet these expectations?
    - Why/Why not?

- If you had complete control, and could change anything to improve your experience of pair programming, what would it be?
    - (Cite common issues reported by students in literature as examples for discussion.)

- All of you have been pair programming for four weeks now. Could each of you tell describe, in your own words, the following roles:
    - Driver
    - Navigator
        - (With regards to good practice, ways to communicate, how to engage your partner.)

- Did you have a particular affinity for one role over the other, or did you switch frequently between both roles?
    - Why/Why not?

Phase 2

- Questions as above, and:
    - If team was exposed to the guidelines:
        - What was your experience with the pair programming guidelines?

        - In your opinion, could the guidelines be used as a taught component to complement your introduction to pair programming?

        - Other comments re: guidelines.

    - If team was not exposed to the guidelines (control group):
        - Following interview, present the guidelines, and ask for initial perceptions and reactions re: usefulness, utility.

Appendix G: Guidelines Evaluation

G1 Forms used for Evaluations with Students

School of Computing — DUNDEE

Information Sheet

The Issue

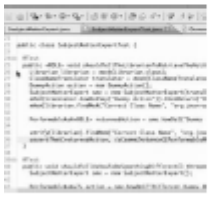
- There is evidence to support the claim that pair programming has certain benefits over more traditional "solo" coding.
- Pilot experiments have shown that 50% of pair programming failures have been attributed to "communication" reasons.
- Little work has been done to study intra-pair communication

Research Proposal

- I am planning to investigate how novice pairs communicate during a pair debugging session, following a semester of pair programming.

Outcomes

- This research can help with:
  - Improving the understanding the scientific community has of communication within pairs
  - Understanding the various roles within the pair
  - Communication guidelines for novices and remote programmers



If you have any further questions, please contact me: (markzarb@computing.dundee.ac.uk). Thank you for your time.

Mark Zarb is a PhD student at the University of Dundee, supervised by Dr Janet Hughes and Prof John Richards.

Images from the pilot investigation at the University of Dundee

School of Computing

Your Involvement

Images from the pilot investigation at the University of Dundee

Thank you for considering helping with this research into further understanding interactions within pairs. This page details the steps required for participation.

Beforehand

- You are asked to give written consent to the project.
  - Participation is completely voluntary. You are asked to give your consent to be observed, allow screen capture and allow audio recording, and optionally, have your sessions video-recorded. Please note that you may withdraw from the research from any time without any penalty, and without stating a reason, with no repercussions. Any data collected shall be treated with full confidentiality, and if published, will not be identifiable as yours. All recordings are not identifiable unless prior written permission is given by you. No undue risk arises from participation in this study.
- We arrange a time and duration for the data collection session(s).
  - This is expected to happen during Week 0 for Semester 2 (07.01 – 11.01).

Observations

- The recordings (screen capture, webcam, dictaphone) are started.
- You complete a questionnaire.
  - This is a short questionnaire on your background as software developers, your working relationship within the pair, and your satisfaction with the pairing session. All questions are completely optional.
- You work as usual for the duration of the session.
  - You are asked to solve as many bugs as you can in the 50-minute time period.

Follow up

- I will provide a debrief detailing any results arising from this data.
- You may be asked to attend an informal interview or a focus group.

If you have any further questions, please contact me: [markzarb@computing.dundee.ac.uk]. Thank you for your time.

Consent Form

Images from the pilot investigation at the University of Dundee

Please read the previous pages carefully, and make sure you understand the procedure. If you are unsure about any of the information presented, feel free to ask the module tutor or your lecturer.

Please note that any observations or footage gathered from these sessions shall only be used for research purposes and will not be viewed by anyone involved in this project other than the researcher until your grades are released..

Optional Consent

This does not affect your eligibility for this study.

• I consent to pictures being taken for research purposes	Yes	No
• I consent to video being taken for research purposes	Yes	No
• I consent to participate in follow-up interviews	Yes	No

Consent

If you agree to participate, please complete this section

• I consent to audio being recorded for research purposes	Yes	No
• I consent to screen capture software being used.	Yes	No
• I agree to the above conditions, and am willing to participate in this study.	Yes	No

- Name: _____
- Signature: _____
- Researcher's Signature: _____
- Date: _____

G2 Survey used for Evaluations with Students (Ch6 Parts 1 & 2)

End-of-session Questionnaire

Any data from this questionnaire used in presentations or other publications will be kept anonymous.

Instructions:	*Please ensure that you read each statement carefully. Complete the items listed overleaf by rating the statement along the scale.*

Please note:

Please answer questions as fully as possible. All answers given are confidential and will not be disclosed to anyone other than authorised researchers at the University of Dundee.

You do not need to answer any or all questions contained in this questionnaire.

Name:

1. I feel pair programming is more beneficial than *solo* programming.

Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree

2. During this session, I found communicating with my partner to be easy.

Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree

3. How long have you been a *solo* programmer?

4. How long have you been pair programming (with other people)?

5. How long have you been pair programming (with today's partner)?

6. Rate your partner's contribution to today's session.

No Participation	Marginal	Satisfactory	Very Good	Excellent

Researcher Use Only

# of Programs Successfully Debugged: ☐

Appendix H: Code-Base for Guidelines Evaluation: Parts 1A & 1B

This appendix provides the code used for the task-based studies described in Chapter 6 of this thesis (Parts 1A and 1B). This study was originally discussed in Murphy et al. (2010); permission to use the code for similar studies was obtained from the authors.

Task 1

```
/**
 * Average.java - calculates the average of three test scores
 *
 */

import java.util.Scanner;

public class Average {

   public static void main(String [] args) {

      int score1, score2, score3;    // test scores
      double average;           // average test score

      Scanner scan = new Scanner(System.in);

      // assume three integer scores are entered
      System.out.print("Enter three test scores:  ");
      score1 = scan.nextInt();
      score2 = scan.nextInt();
      score3 = scan.nextInt();

      //determine the average
      average = (score1 + score2 + score3) / 3;

      System.out.println("Average:  " + average);
   }
```

Task 2

```
/**
 * Volume.java - calculates the volume of a cube
 *
 */

import java.util.Scanner;

public class Volume{

   public static void main(String [] args) {

      double side;   // side of the cube
      double volume; // volume of the cube

      Scanner scan = new Scanner(System.in);

      // assume a positive numeric value is entered for the side length
      System.out.print("Enter the length of a side of the cube: ");
      side = scan.nextDouble();

             // calculate volume
      volume = (Math.pow(3, side));

      System.out.println("Volume of the cube: " + volume);

    }

}
```

Task 3

```
/**
 *  Rectangle1.java - draws a rectangle
 *
 * Inputs two integers, n and m, and outputs an n x m rectangle.
 * For example, for 4 and 7:

  *******
  *******
  *******
  *******

 **/

import java.util.Scanner;
public class Rectangle1
{
       public static void main(String [] args)
       {
          int n, m;
          Scanner scan = new Scanner(System.in);

          // assume values entered for n and m will be positive integers
          System.out.print("Enter the number of rows: ");
          n = scan.nextInt();
          System.out.print("Enter the number of columns: ");
          m = scan.nextInt();

              // draws solid rectangle
              for (int r = 1; r <= m; r++)
              {
                     for (int c = 1; c <= n; c++)
                            System.out.print("*");
                     System.out.println();
              }

              System.out.println();

       }
} // end of class Rectangle1
```

Task 4

```
/**
 *  Rectangle2.java - draws a hollow rectangle
 *
 * Inputs two integers, n and m, and outputs an n x m hollow rectangle.
 * For example, for 4 and 7:

  *******
  *     *
  *     *
  *******

 **/

import java.util.Scanner;
public class Rectangle2
{
       public static void main(String [] args)
       {
          int n, m;
          Scanner scan = new Scanner(System.in);

          // assume values entered for n and m will be positive integers
          System.out.print("Enter the number of rows: ");
          n = scan.nextInt();
          System.out.print("Enter the number of columns: ");
          m = scan.nextInt();

              // draws top row of hollow rectangle
              for (int c = 0; c <= m; c++)
                     System.out.print("*");
              System.out.println();

              // draws inner rows of hollow rectangle
              for (int r = 2; r < n; r++)
              {
                     System.out.print("*");
                     for (int c = 1; c < m; c++)
                            System.out.print(" ");
                     System.out.println("*");
              }

              // draws final row of hollow rectangle
              for (int c = 0; c <= m; c++)
                     System.out.print("*");
              System.out.println();


              System.out.println();

       }
} // end of class Rectangle2
```

Task 5

```
/**
 * Validate.java - reads 20 quiz scores and confirms that they are in the
range 0-10
 *
 */

import java.util.Scanner;

public class Validate {

   public static void main(String [] args) {

      int quiz;    //integer value

      Scanner scan = new Scanner(System.in);

      for (int count = 0; count < 20; count++) {

             // assume an integer quiz score is entered
             System.out.print("Enter a quiz score between 0 and 10:  ");
             quiz = scan.nextInt();

             if (quiz > 0 || quiz < 10) {
                    System.out.println(quiz + " is valid.");
             }
             else {
                    System.out.println(quiz + " is invalid.");
             }
      }
   }
}
```

Task 6

```
/**
 *  Raffle1.java - calculates the student average and total ticket sales for
 *  a school bike raffle
 *
 */

import java.util.Scanner;

public class Raffle1
{
       public static void main(String[] args)
       {
              double ticketPrice;
              int numChildren, ticketsSold, totalTickets;
              String studentName;

              Scanner scan = new Scanner(System.in);

              // assume a non-negative numeric value is entered for ticket
price
              System.out.print("Enter the ticket price: ");
              ticketPrice = Double.parseDouble(scan.nextLine());

              numChildren = 0;
              totalTickets = 0;

              // assume only the student's first name is entered
              System.out.println("\nEnter the name and tickets sold for each
child (\"stop\" to quit): ");
              studentName = scan.next();

              while(studentName != "stop")
              {
                     // assume an integer value is entered for ticketsSold
                     ticketsSold = scan.nextInt();
                     scan.nextLine(); // consumes '\n'
                     numChildren++;
                     totalTickets += ticketsSold;
                     System.out.println (studentName + " sold $" + ticketsSold
* ticketPrice +
                            " worth of tickets\n");
                     studentName = scan.next();
              }

              System.out.println("Average number of tickets sold per child: " +
(double) totalTickets/numChildren);
              System.out.println("The class sold " + totalTickets + " tickets
worth: $" + totalTickets * ticketPrice);
    } // end of main
} // end of Raffle1
```

Task 7

```
/**
 *  Rectangle3.java - draws a checkered rectangle
 *
 * Inputs two integers, n and m, and outputs a checkered rectangle.
 * For example, for 5 and 9:


  * * * * *
   * * * *
  * * * * *
   * * * *
  * * * * *

 **/

import java.util.Scanner;
public class Rectangle3
{
       public static void main(String [] args)
       {
          int n, m;
          Scanner scan = new Scanner(System.in);

          // assume values entered for n and m will be positive integers
          System.out.print("Enter the number of rows: ");
          n = scan.nextInt();
          System.out.print("Enter the number of columns: ");
          m = scan.nextInt();

         // draws checkerd rectangle
         for (int r = 1; r <= n; r++)
         {
              for (int c = 1; c <= m; c++)
                     if (c % 2 == 0)
                            System.out.print("*");
                     else
                            System.out.print(" ");
              System.out.println();
          }

       }
} // end of class Rectangle3
```

Task 8

```
/**
 * Validate2.java - reads students' grades and determines if they are between
0.0 and 4.0
 *
 */

import java.util.Scanner;

public class Validate2 {

   public static void main(String [] args) {

      double grade;    //grade input
      String name;

      Scanner scan = new Scanner(System.in);

      // assume the word "quit" (not q or Q) is entered to stop
      System.out.print("\nEnter a student's name (enter \"quit\" when you're
done): ");
      name = scan.nextLine();

      while (!name.equalsIgnoreCase("quit")) {

             // assume a numeric grade is entered
             System.out.print("Enter " + name +"'s decimal grade: ");
             grade = scan.nextDouble();
             scan.nextLine(); // consumes '\n'

            if (grade < 0.0 && grade > 4.0) {
                  System.out.println(name + "'s " + grade + " grade is not
valid.");
            }
            else {
                  System.out.println(name + "'s " + grade + " is a valid
grade.");
            }

           System.out.print("\nEnter a student's name (enter \"quit\" when
you're done): ");
           name = scan.nextLine();
      }
   }
}
```

Task 9

```
/**
 * Sort3.java - reads three integers and displays them in ascending order
 *
 */

import java.util.Scanner;

public class Sort3Integers {

   public static void main(String [] args) {

      int num1, num2, num3;    // numbers to sort

      Scanner scan = new Scanner(System.in);

      // assume integer values are entered for num1, num2 and num3
      System.out.print("Enter three integers:  ");

      num1 = scan.nextInt();
      num2 = scan.nextInt();
      num3 = scan.nextInt();

      // order the nums so that num1 is the smallest, then num2, then num3 and
print
          int temp;
          if (num3 < num2) {
             temp = num2;
             num2 = num3;
             num3 = temp;
          }
          if (num2 < num1) {
             temp = num1;
             num1= num2;
             num2= temp;
          }
          if (num2 < num3) {
             temp = num2;
             num2= num3;
             num3 = temp;
          }

         System.out.println("The numbers sorted:   " + num1 + "   " + num2 + "
" + num3);

   }
}
```

Task 10

```
/**
 *  Raffle2.java - calculates statistics for a school bike raffle
 *
 */

import java.util.Scanner;

public class Raffle2
{
	public static void main(String[] args)
	{
		double ticketPrice;
		int numChildren, ticketsSold, totalTickets, maxSold;
		String studentName, maxName;

		Scanner scan = new Scanner(System.in);

		// assume a non-negative numeric value is entered for the ticket
price
		System.out.print("Enter the ticket price: ");
		ticketPrice = Double.parseDouble(scan.nextLine());

		numChildren = 0;
		totalTickets = 0;
		maxSold = 0;
		maxName = "";

		// assume name and tickets sold input is correctly formatted
		// and the word "stop" is entered to quit
		System.out.println("\nEnter the name and tickets sold for each
child (\"stop\" to quit): ");

		studentName = scan.next();

		while( !studentName.equalsIgnoreCase("stop") )
		{
			ticketsSold = scan.nextInt();
			scan.nextLine(); // consumes '\n'
			numChildren++;
			totalTickets += ticketsSold;
			if (ticketsSold > maxSold)
				maxSold = ticketsSold;
				maxName = studentName;
			studentName = scan.next();
		}

		System.out.println("Average number of tickets sold per child: " +
(double) totalTickets/numChildren);
		System.out.println("Most tickets sold by one child: " + maxSold +
" by " + maxName);
		System.out.println("The class sold " + totalTickets + " tickets
worth: $" + totalTickets * ticketPrice);
	} // end of main
} // end of RaffleBuggy
```

Task 11

```
// class Car to hold information about an automobile

public class Car
{
	String make, model;
	double mpg;

	// constructor for Car
	public Car(String mk, String mdl)
	{
		make = mk;
		model = mdl;
		mpg = 0.0;
	}

	// returns the car's make
	public String getMake()
	{
		return make;
	}

	// returns the car's model
	public String getModel()
	{
		return model;
	}

	// calculates mpg given miles and gallons
	public void calcMpg(int miles, int gallons)
	{
		double mpg = (double) miles / gallons;
	}

	// returns mpg
	public double getMpg()
	{
		return mpg;
	}

	// main method to test Car
	public static void main(String[] args)
	{
		Car myCar = new Car("Honda", "CRV");
		myCar.calcMpg(245, 10);
		System.out.println("My    "+    myCar.getMake()    +    "    "    +
myCar.getModel() +
		 " gets " + myCar.getMpg() + " miles to the gallon.");
	}
}
```

Task 12

```
/**
 * TriangleType.java - determines triangle type given three side lengths
 *
 */

import java.util.Scanner;

public class TriangleType {

   public static void main(String [] args) {

      int side1, side2, side3;   // sides of the triangle

      Scanner scan = new Scanner(System.in);

      // assume positive integer values are entered for the three sides
      System.out.print("Enter three numbers to form a triangle: ");

      side1 = scan.nextInt();
      side2 = scan.nextInt();
      side3 = scan.nextInt();

          int temp;
          if (side3 < side2) {
             temp = side2;
             side2 = side3;
             side3 = temp;
          }
          if (side2 < side1) {
             temp = side1;
             side1= side2;
             side2= temp;
          }
          if (side3 < side2) {
             temp = side2;
             side2= side3;
             side3 = temp;
          }

         System.out.println("Sides sorted:  " + side1 + "  " + side2 + "  " +
side3);

         // figure out the kind of triangle (based on side lengths) and print
         System.out.print("Triangle Type:  ");
          if (side1 + side2 <= side3) {
             System.out.println("DOES NOT FORM TRIANGLE");
          }
          else if (side1 == side3)
             System.out.println("ISOSCELES");
          else if ((side1 == side2) || (side2== side3))
             System.out.println("EQUILATERAL");
          else
             System.out.println("SCALENE");

         System.out.println();

    }

}
```

Task 13

```
/* Search.java -- generates an array of 20 random integers, prints them,
 *     then finds the position of an element specified by the user.
*/

import java.util.Random;
import java.util.Scanner;

public class Search {

	public static void main (String[] args) {

		int[] numbers = new int[20];
		int searchValue ; // value to search for
		int position ;  // position of the element in the array

		Scanner keyboard = new Scanner(System.in);

		fillArray( numbers );
		System.out.print("Array: ");
		printArray( numbers );

		// assumes an integer value is entered
		System.out.print("Enter a value to search for: ");
		searchValue = keyboard.nextInt();
		position = search(searchValue, numbers);
		if (position != -1)
			System.out.println("The value " + searchValue + " is in
position " + position);
		else
			System.out.println("The value " + searchValue + " is not
in the array ");
	}

	// fills an array with random numbers between 1 and 100
	public static void fillArray(int[] numbers) {

		Random rand = new Random();
		for (int i = 0; i < numbers.length; i++)
			numbers[i] = rand.nextInt(100) + 1;

	}

	// prints the contents of an array to the terminal
	public static void printArray(int[] numbers) {

		System.out.println();

		for (int i = 0; i < numbers.length; i++)
			System.out.print( numbers[i] + " " );

		System.out.println('\n');
	}

	// returns the position of the first occurrence of a value
	// in an array of ints and -1 if the value is not found
	public static int search(int searchValue, int[] numbers) {

		int i = 0, position = -1;
		while( numbers[i] != searchValue ) {
			i++;
			if (numbers[i] == searchValue)
				position = i;
```

```
            }

            return position;
        }
}
```

Task 14

```
/**
 * Calculator1.java - implements a simple infix calculator
 *
 * This program implements a very simple calculator that multiplies, adds
 * and subtracts. It accepts expressions like +13+4*5= and prints the
 * result. Each integer or operation is entered on a separate line
 * and should not include precedence or brackets. Sample execution:

Enter your expression (start with a '+', type '=' when you want answer):
+
13
+
2
*
4
=
The answer is 60.0

 */

import java.util.Scanner;
public class Calculator1 {

    public static void main(String[] args){
      int answer;
      String currentlyRead;
      int opnd1,opnd2;
      char op;

      Scanner scan = new Scanner(System.in);

      answer=0;

      // assume a correctly formatted expression is entered by the user
      System.out.print("Enter your expression, start with a '+' ");
       System.out.println("type '=' when you want answer:");
      currentlyRead=scan.nextLine();
      op=currentlyRead.charAt(0);
      while (!currentlyRead.equals("=")) {
         opnd2= scan.nextInt();
         scan.nextLine(); // advance to next line
          if (op=='+')
              answer=answer+opnd2;
          else if (op=='-')
              answer=answer-opnd2;
          else if (op=='*')
                answer=answer*opnd2;
          else {
             System.out.println("invalid operation");
             System.exit(1);
          }
          currentlyRead=scan.nextLine();
       }
       System.out.println("The answer is "+answer);
    }
}
```

Task 15

```
/**
 *  Raffle3.java - calculates statistics for a school bike raffle
 *
 */

import java.util.Scanner;

public class Raffle3
{
	public static void main(String[] args)
	{
		double bikeCost, overheadCost, ticketPrice;
		int numChildren, ticketsSold, totalTickets;
		String studentName;

		Scanner scan = new Scanner(System.in);

		// assume only numeric values are entered
		System.out.print("Enter the cost of the bike: ");
		bikeCost = Double.parseDouble(scan.nextLine());
		System.out.print("Enter any overhead costs (e.g., printing,
incentives): ");
		overheadCost = Double.parseDouble(scan.nextLine());
		System.out.print("Enter the ticket price: ");
		ticketPrice = Double.parseDouble(scan.nextLine());

		numChildren = 0;
		totalTickets = 0;

		// assume the word "stop" (not s or S) are entered to quit
		System.out.println("\nEnter the name and tickets sold for each
child (\"stop\" to quit): ");

		studentName = scan.next();

		while( !studentName.equalsIgnoreCase("stop") )
		{
			// assume an integer value is entered for ticketsSold
			ticketsSold = scan.nextInt();
			scan.nextLine(); // consumes '\n'
			numChildren++;
			totalTickets += ticketsSold;
			studentName = scan.next();
		}

		System.out.println("\nTo break even you should have sold at least
" +
			Math.ceil(bikeCost + overheadCost / ticketPrice) + "
tickets ");

		System.out.println("Average number of tickets sold per child: " +
(double) totalTickets/numChildren);
		System.out.println("The class sold " + totalTickets + " tickets
worth: $" + totalTickets * ticketPrice);
		System.out.println("Total profit from the raffle: $" +
(totalTickets * ticketPrice - bikeCost - overheadCost));

	} // end of main
} // end of Raffle3
```

Task 16

```
/* FindSmallest.java -- generates an array of 20 random integers, prints them,
 *     then finds and displays the smallest value in the array.
*/

import java.util.Random;

public class FindSmallest
{

	public static void main (String[] args) {

		int[] numbers = new int[20];

		fillArray( numbers );
		System.out.print("Array: ");
		printArray( numbers );
		System.out.println("The smallest element in the array is: " +
			findSmallest( numbers ));
	}

	// fills an array with random numbers between 1 and 100
	public static void fillArray(int[] numbers) {

		Random rand = new Random();

		for (int i = 0; i < numbers.length; i++)
			numbers[i] = rand.nextInt(100) + 1;

	}

	// prints the contents of an array to the terminal
	public static void printArray(int[] numbers) {

		System.out.println();

		for (int i = 0; i < numbers.length; i++)
			System.out.print( numbers[i] + " " );

		System.out.println('\n');
	}

	// reverses the contents of an array of ints
	public static int findSmallest(int[] numbers) {

		int small = numbers[0];

		for (int i = 1; i < numbers.length; i++)
		{
			if (numbers[i] < small)
				return numbers[i];
		}

		return small;
	}
}
```

Task 17

```
/**
 *  Calculator2.java  -  implements  a  simple  infix  calculator  that  allows
multiple
 * expressions. For example:

Do you want to use the calculator- Yes or No? Yes
Enter your expression (start with a '+', type '=' when you want answer):
+
13
+
2
*
4
=
The answer is 60.0

Do you want to use the calculator- Yes or No? Yes
Enter your expression (start with a '+', type '=' when you want answer):
+
13
-
2
=
The answer is 11.0

Do you want to use the calculator- Yes or No? No


*/

import java.util.Scanner;
public class Calculator2 {

    public static void main(String[] args){
      int answer;
      String currentlyRead;
      int opnd1,opnd2;
      char op;

      Scanner scan = new Scanner(System.in);

      // assume the user enters the word "Yes" (not y or Y) to continue
      System.out.println("Do you want to use the calculator- Yes or No?");
      String response= scan.nextLine();
      answer=0;
      while (response.equalsIgnoreCase("Yes")){

       // assume a correctly formatted expression is entered by the user
        System.out.print("Enter your expression, start with a '+' ");
               System.out.println("type '=' when you want answer:");
        currentlyRead=scan.nextLine();
        while (!currentlyRead.equals("=")) {
           op=currentlyRead.charAt(0);
            opnd2= scan.nextInt();
            scan.nextLine(); // advance to next line
            if (op=='+')
                answer=answer+opnd2;
            else if (op=='-')
                answer=answer-opnd2;
            else if (op=='*')
                answer=answer*opnd2;
            else {
```

```
                System.out.println("invalid operation");
                System.exit(1);
            }
            currentlyRead=scan.nextLine();
        }
     System.out.println("The answer is "+answer);
     System.out.println("Do you want to use the calculator- Yes or No?");
     response= scan.nextLine();
     }
    }
}
```

Task 18

```
/**
 * Sort3.java - sorts three integers in ascending order
 *
 */

import java.util.Scanner;

public class Sort3 {

   public static void main(String [] args) {

      int x, y, z;

      Scanner scan = new Scanner(System.in);

      // assume three integer values are entered
      System.out.print("Enter three integer values: ");
      x = scan.nextInt();
      y = scan.nextInt();
      z = scan.nextInt();

      System.out.println("\nBefore sort: x = " + x + " y = " + y + " z = " +
z);

      if (z < y) {
              swap(y, z);
      }
      if (y < x) {
              swap(x, y);
      }
      if (z < y) {
              swap(y, z);
      }

      System.out.println("\nAfter sort:  x = " + x + " y = " + y + " z = " +
z);

    }

    public static void swap(int x, int y)
    {
             int temp = x;
             x = y;
             y = temp;
       }

}
```

Task 19

```
/* Reverse.java -- generates an array of 20 random integers, prints them,
 *     reverses the array, and prints them again.
*/

import java.util.Random;

public class Reverse
{

	public static void main (String[] args) {

		int[] numbers = new int[20];

		fillArray( numbers );
		System.out.print("Original array: ");
		printArray( numbers );
		reverseArray( numbers );
		System.out.print("Reversed array: ");
		printArray( numbers );
	}

	// fills an array with random numbers between 1 and 100
	public static void fillArray(int[] numbers) {

		Random rand = new Random();

		for (int i = 0; i < numbers.length; i++)
			numbers[i] = rand.nextInt(100) + 1;

	}

	// prints the contents of an array to the terminal
	public static void printArray(int[] numbers) {

		System.out.println();

		for (int i = 0; i < numbers.length; i++)
			System.out.print( numbers[i] + " " );

		System.out.println('\n');
	}

	// reverses the contents of an array of ints
	public static void reverseArray(int[] numbers) {

		int temp;

		for (int i = 0; i < numbers.length; i++)
		{
			temp = numbers[i];
			numbers[i] = numbers[numbers.length-1-i];
			numbers[numbers.length-1-i] = temp;
		}
	}
}
```

Appendix I: Surveys for Guidelines Evaluation: Part 2

This appendix provides the instruction sheet and surveys that were provided to all students who participated in Part 2 of the study reported in Chapter 6.

I1 Instruction Sheet

Pair Programming Study III

An Introduction

You can control your character in NetBeans using a series of commands.

This is an example of the main class in the project:

```
public static void main(String args[]) {

    Game main = new Game();
```

//This line tells the game which map to load. You will need to change this based on the task you are asked to complete.

```
    main.setMap("blank");

    main.showGame();
```

// Any code you write goes here.

```
}
```

The APE tool uses Java – you can use any kind of code (e.g. for, while, if loops, etc.) in order to complete the maps. The following code is required to make your character move:

```
    main.move();
```

//This command moves your character one space forward in the direction faced.

```
    main.turnLeft();
    main.turnRight();
```

//These commands tell your character to turn 90 degrees clockwise, or anti-clockwise.

The Maps

Please complete as many of these maps as you can. At the bottom of the screen, you will see a prompt showing you how many steps your character has taken: please write down these steps for each map you complete on this sheet.

When you complete each map, please copy the map code into a text file and save it under your folder.

`main.setMap("_____");`
`bimble`
`central_cavern`
`dizzy`
`you_will_blink_first`
`first_steps`
`ghost_train`
`killing_time`
`you_make_me_feel_like_dancing`
`i_smell_your_fear`

I2 Post-Study Survey

Post-Study Survey

1. I feel pair programming is more beneficial than solo programming.

Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree

2. During this session, I found communicating with my partner to be easy.

Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree

3. Rate your pair partner's contribution to today's session.

No Participation	Marginal	Satisfactory	Very Good	Excellent

4. How long have you been a: i) solo programmer; ii) pair programmer?

Solo Programmer	Pair Programmer

5. What role do you feel you predominantly played in today's session?

Driver	Navigator

Appendix J: Industry Feedback

This appendix provides a transcribed copy of comments made on the guidelines, provided by the industry members that were observed in Chapters 3 and 4.

Comments on *Restarting*

- I would also suggest walking away from the pairing desk and taking regular breaks (for tea, coffee, etc.)
- Agree: depending on the depth of the work, the review stage might consist of trying to decompose a particular unit test into smaller steps (if the block is around an implementation problem), or defining an acceptance test to "thrash out" the specific problem.
- I like this style (it is similar to Pomodoro). But make sure there are not too many "unfocusing" points. Need some focus.
- Give him some space to read the code himself [before suggesting next steps].
- Giving voice to the thoughts might help.
- Careful with [breaking focus] – sometimes too much interruption can be harmful. A good balance is the key.
- Good advice!!
- I agree but when you're trying to think something through there is social 'pressure' to continue to talk when thinking quietly could be more useful. We almost never "go silent"! But when stuck often chat, get coffee, etc.
- Pairing works best when you have a reasonable idea how to solve the task at hand. Exploring code or intense concentration required for problem solving may

be better done alone. If stuck on a small problem then writing a test provides a new way of thinking about the problem.

- Generally helpful to break for coffee, go to lunch, etc. when we're stuck. "Stuck" usually means "can't agree which of various approaches to take" rather than "can't think of anything". So *stuck* is not a silent thing!
- …or go for a coffee! Sometimes bringing in a third person and talking through the current thought process or where the pair is stuck helps kick-start a fresh thought process.
- Don't dismiss your partner trying to break focus.
- I agree [with breaking focus], it can be tiring – this helps fight fatigue and also can break out of a rut.
- Snacking/drinking at these times is nice too. Taking a break just after writing a failing test can be beneficial, so when you get back to work, you know where to continue. Plus, your partner may want a break. Without the break, he might not be able to work well.
- Suggesting next steps helps to avoid over-engineering (e.g. trying to make the solution more generalised, to cope with requirements that we don't need or understand yet).
- Agree – also look for other examples – and try to take advantage of your partner's experience.
- True [re: breaking focus] – but not THAT often. It's easy to lose focus. I find useful also to have a coffee break with my partner. Looking at hard tasks while drinking a good coffee and taking a small walk can be really helpful.

Comments on *Planning*

- Agree.
- I feel TDD, and alternating the keyboard after each test implementation combo keeps both partners in sync.
- …i.e. learning to say *I don't know* or *I don't understand* is critical.
- Agree re: clarification.
- We typically *do* review and explain at check-in, especially when this is the first activity of a pair with a new member.
- Fits in with TDD (write test, pass test, refactor). Each phase provides an opportunity to switch the driver (hand over keyboard).
- Again totally agree with benefits of discussion, clarifying motivation, etc. But interruption can derail thought processes, which is challenging. Can be very useful, and avoid mistakes though.
- This [offering an explanation of the current state] does help many times, mostly to realise that you're on the wrong track.
- Depending on people's memory I'd add a subtask to suggesting where a bullet point is scribbled down (informally we refer to this as a shit list) to help track multiple paths. Some problems do not always have clear pro/cons for a particular implementation.
- It is important to capture discussed (and agreed) suggestions and reviews so that they do not get lost and so that similar discussions are not unnecessarily repeated over and over. Typically, you would capture these in the form of tech tasks to be added to the project's backlog.

- Another benefit is to minimise disruptions/going off-tangent from the task at hand.

Comments on *Action*

- This is good. I have had some silent partners and it tends to cause frustration as unless you know the pair very well and/or the problem domain, silent partners just look like they're clicking randomly on the screen.
- Switch regularly between roles (keeps both members sharp and involved).
- Really useful – best part of PP, I think.
- Agree – but sometimes, you need to type and explain afterwards.
- Agree. This helps navigator know things that might have been overloaded, avoid suggesting things that the driver was about to do, and stop the navigator from interrupting a train of thought.
- The driver should articulate what he is doing and thinking, not mutter.
- I don't think muttering from either the driver or the navigator is a good thing, as the driver should voice the thoughts as they drive.
- I'm rarely comfortable with the driver/navigator pattern, although for some it seems to validate asymmetrical interactions (which seem okay and not to need validation to me).
- I tend to leave suggestions until the "refactor" part of TDD.
- [Voicing your thoughts] helps your partner not get bored/distracted too.
- This is all true. Also think about what the current test is not covering? Is there anything left out that is worth verifying?
- True. Also I used to prioritise the Navigator's issues.